Design and Scheduling of Efficient Real-Time Embedded Systems

Zaid Al-bayati

Department of Electrical and Computer Engineering
McGill University
Montréal, Québec, Canada

January 2017

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

To my mother, my father, Ruaa, Mustafa, and Mariam

Abstract

Computer systems have gone through tremendous changes in the past few decades. Relatively large general purpose computers dominated the early days of computers. With time, demand increased for smaller, more dedicated computer systems, called *embedded systems*. These systems perform a specific set of functions interacting with the physical environment, often in real-time. Real-time embedded systems are found today in many application domains such as the automotive domain, avionics, and control systems. Real-time systems differ from traditional computer systems in their dependence on time as a correctness criteria, i.e., a late correct answer is useless for these systems.

Embedded real-time systems today are more integrated, more parallel, and more complex than ever before. In this thesis, we discuss limitations that affect the applicability of real-time models, analysis methods, and scheduling approaches to the realities of today's embedded systems and propose solutions to address these challenges.

We first look into the issue of shared resources and its effect on the mapping and scheduling of software tasks in a real-time system. Most task mapping approaches proposed in the literature perform task mapping assuming independent tasks that do not share resources. Managing shared resources and their protection mechanisms is performed later. However, this approach might require several rounds of iteration and can lead to inefficient results. In this thesis, we explore the possibility of using different resource protection mechanisms within a single system, and propose to tackle the design problem more efficiently by jointly performing task allocation, scheduling, and resource pro-

tection mechanism selection. Two approaches are presented to solve this optimization problem: an optimal Mixed Integer Linear Programming (MILP) approach and an efficient heuristic. The proposed work is shown to significantly improve system schedulability. Experimental results indicate that the minimum utilization at which at least 95% of systems become scheulable can be improved from 65%–70% for the best published task allocation algorithms to 76%–85% using our heuristic with minimal memory cost. Even better results can be achieved using the MILP approach.

Next, we look into the design of systems composed of components that have different levels of criticality. *Mixed-Criticality Systems* (MCS) received much attention recently to due their industrial relevance. We focus on three challenges in MCS design: task allocation, fault-tolerance, and model-based design. For task allocation, we show that traditional task allocation algorithms can be inefficient in a mixed-criticality context, and propose an alternative that we call *dual-partitioned* task allocation. Experiments show that for systems that have a utilization of 80% or higher, we can schedule 17% more systems on a given multicore platform using the dual-partitioned approach.

Fault-tolerance is an important issue for MCS since these systems contain a safety critical part. To design MCS that tolerate hardware transient faults, we propose a new mixed-criticality model that simultaneously addresses criticality, reliability, and Quality of Service (QoS). A schedulability test for the new model is derived. Furthermore, to allow designers to incorporate the new model and analysis in their design process, we propose a design space exploration framework based on the new model that supports various fault-tolerance

mechanisms. QoS improvements of up to 42.9% can be achieved using the new model compared to the traditional MCS model extended to support transient faults. To overcome more serious faults that can cause a processor in the system to fail, we present a new design approach for MCS. The standard MCS model operation and analysis are extended to cover this failure scenario. Mapping and scheduling is performed using a new MILP formulation. Experiments show that the proposed approach achieves a 3.2X increase in the number of schedulable systems compared to a baseline design process.

Model-based design is used in many domains for designing complex systems starting from high level models. We focus specifically here on the Synchronous Reactive (SR) model since it is widely used for control-dominated applications. A major challenge when systems designed using the SR model are implemented is the preservation of the model semantics. Sometimes, the model semantics can not preserved unless blocks known as functional delays are added to the system, with negative effects on system performance. In this thesis, we propose algorithms to generate optimized semantic-preserving implementations for MCS specified using the SR model, with minimal functional delay addition. An optimal Branch-and-Bound based algorithm and an efficient heuristic are proposed for this purpose.

Abrégé

Les systèmes informatiques ont subi des changements énormes au cours des dernières décennies. Dans leurs débuts, les ordinateurs, de grande taille et à usage général, étaient dominants. Avec le temps, la demande pour des systèmes informatiques plus petits et dédiés pour des tâches plus spécifiques, appelés *systèmes embarqués*, a augmenté. Ces systèmes exécutent un ensemble de fonctions spécifiques interagissant avec l'environnement physique, souvent en temps réel. Les systèmes embarqués temps-réel se trouvent aujourd'hui dans de nombreux domaines d'application tels que l'automobile, l'avionique et les systèmes de contrôle. Les systèmes temps-réel diffèrent des systèmes informatiques traditionnels dans leur dépendence au temps qui est utilisé comme critère de correction. C'est-à-dire qu'une réponse correcte tardive est inutile pour ces systèmes.

Les systèmes embarqués temps-réel sont aujourd'hui plus intégrés, plus parallèles et plus complexes que jamais. Dans cette thèse, nous discutons des limites qui affectent l'applicabilité des modèles temps-réel, des méthodes d'analyse et des approches d'ordonnancement aux réalités des systèmes embarqués d'aujourd'hui et nous proposons des solutions pour relever ces défis.

En premier lieu, nous examinons la question des ressources partagées et leurs effets sur la cartographie et l'ordonnancement des tâches logicielles dans un système temps-réel. La plupart des approches de cartographie des tâches proposées dans la littérature effectuent la cartographie des tâches en assumant des tâches indépendantes qui ne partagent pas les ressources. La gestion des ressources partagées ainsi que leurs mécanismes de protection sont ef-

fectués plus tard. Cependant, cette approche peut nécessiter plusieurs cycles d'itération et peut mener à des résultats inefficaces. Dans cette thèse, nous explorons la possibilité d'utiliser différents mécanismes de protection des ressources au sein d'un même système et proposons d'aborder plus efficacement le problème de conception en exécutant conjointement l'attribution des tâches, l'ordonnancement et la sélection des mécanismes de protection des ressources. Deux approches sont présentées pour résoudre ce problème d'optimisation: une approche doptimisation linéaire à nombres entiers mixtes optimale (MILP) et une heuristique efficace. Le travail proposé permet d'améliorer considérablement lordonnançabilité du système. Les résultats expérimentaux indiquent que l'utilisation minimale à laquelle au moins 95% des systèmes deviennent ordonnançables peut être améliorée de 65%–70%, dans les meilleurs algorithmes d'allocation de tâche publiés, à 76%–85% en utilisant notre heuristique avec un cot mémoire minime. Des résultats encore meilleurs peuvent être obtenus en utilisant l'approche MILP.

Ensuite, nous examinons la conception de systèmes formés de composants qui ont différents niveaux de criticité. Les *systèmes de criticité mixte* (MCS) ont récemment reçu beaucoup d'attention en raison de leur pertinence industrielle. Nous nous concentrons sur trois défis en matière de conception MCS: lattribution des tâches, la tolérance aux pannes et la conception basée sur modèle. Pour l'attribution des tâches, nous montrons que les algorithms traditionnels, d'allocation de tâches peuvent être inefficaces dans un contexte de criticité mixte et proposer une alternative que nous appelons allocation de tâches à *double partition*. Les expériences montrent que pour les systèmes

dont l'utilisation est supérieure ou égale à 80%, qu'on peut ordonnancer 17% de systèmes sur une plate-forme multicœur donnée en utilisant l'approche à deux partitions.

La tolérance aux pannes est un problème important pour les MCS puisque ces systèmes contiennent une partie critique pour la sécurité. Pour concevoir des MCS qui tolèrent les défauts transitoires, nous proposons un nouveau modèle de criticité mixte qui aborde simultanément la criticité, la fiabilité et la qualité de service (QoS). Un test d'ordonnançabilité pour le nouveau modèle est déduit. En outre, pour permettre aux concepteurs d'intégrer le nouveau modèle et l'analyse dans leur processus de conception, nous proposons un cadre d'exploration d'espace de conception basé sur le nouveau modèle qui prend en charge divers mécanismes de tolérance de pannes. Des améliorations de la qualité de service jusqu'à 42.9% peuvent être obtenues en utilisant le nouveau modèle comparé au modèle MCS traditionnel qui peuvent supporter les pannes transitoires. Pour surmonter les pannes plus graves qui peuvent entraner l'échec d'un processeur dans le système, nous présentons une nouvelle approche de conception pour MCS. Le fonctionnement et l'analyse du modèle MCS standard sont élargis pour couvrir ce scénario de défaillance. La cartographie et l'ordonnancement sont effectuées en utilisant une nouvelle formulation MILP. Les expériences montrent que l'approche proposée permet d'obtenir une augmentation de 3.2 fois du nombre de systèmes ordonnançables comparé à un processus de conception de base.

La conception basée sur modèles est utilisée dans de nombreux domaines pour concevoir des systèmes complexes à partir de modèles de haut niveau.

Nous focalisons, ici, spécifiquement sur le modèle réactif synchrone (SR), car il est largement utilisé pour les applications dominées par les systèmes de contrôle. Un défi majeur lorsque les systèmes conçus à l'aide du modèle SR sont implementés est la préservation du modèle sémantique. Parfois, le modèle sémantique ne peut pas être conservé à moins que des blocs connus comme des retards fonctionnels soient ajoutés au système, avec des effets négatifs sur la performance du système. Dans cette thèse, nous proposons des algorithmes pour générer des implémentations de conservation sémantique optimisées pour MCS spécifiées à l'aide du modèle SR, avec un minimum d'ajout de retard fonctionnel. Un algorithme optimal basé sur Branch-and-Bound et une heuristique efficace sont proposés à cet effet.

Acknowledgements

Although the cover of this document bears my name alone, the reality is far from that. The work in this thesis would not have been possible without the contribution and support of my supervisors, family, and colleagues whom I owe alot.

First of all, I would like to thank my supervisors: Dr. Haibo Zeng and Dr. Brett Meyer. I am deeply grateful to you for your guidance, advice, continuous support, and encouragement. I learned a lot from you throughout this journey as supervisors, as teachers and as persons. I am thankful for the countless hours you contributed to this work and for your support when things were not going so well. I am glad to have worked with you and will always be indebted to you.

I would like also to thank members of the supervising committee: Dr. Zeljko Zilic and Dr. Jörg Kienzle for their insightful comments and feedback which helped improve this work.

I am also thankful to Dr. Marco Di Natale for his guidance and insights while I was working on multicore resource sharing. I would also like to thank Qingling Zhao for her contribution and collaboration on mixed-criticality systems, Jonah Caplan for his contributions to fault-tolerance in mixed-criticality systems, and Dr. Zonghua Gu for his contribution to model-based design in mixed-criticality systems. Many thank to my colleagues at the Reliable Silicon System Lab (RSSL) at McGill University for your support and insightful discussions.

I am also grateful to the Fonds de Recherche du Québec - Nature et Tech-

nologies (FRQNT) and to McGill University for financially supporting this work.

This work would not have been remotely possible without the support of my family. First, my mother, whom no word in the English language can justly thank enough for her support and prayers throughout this journey. My father, who has always supported and encouraged me. My fiancée, Ruaa, who has been a constant source of support and encouragement, and has been patient with my absence. My brother, Dr. Mustafa, my source of strength. My sister, Mariam, the brightest product of the family. I am deeply grateful to you.

Contents

1 Introduction **1**

1.1 Timing in Embedded System Design 2

1.2 Motivation 4

1.3 Thesis Statement 8

1.4 Thesis Contributions 9

1.4.1 Mapping and scheduling in resource-sharing multicore systems 9

1.4.2 Task mapping in multicore mixed-criticality systems . . . 10

1.4.3 Fault-tolerance in mixed-criticality systems 11

1.4.4 Implementing synchronous reactive models of mixed-criticality systems 12

1.5 Related Publications 12

1.6 Thesis Outline 16

2 Background and Related Work **17**

2.1 System Model: Basic Definitions 17

2.2 Scheduling 20

2.2.1 Scheduling in multicore architectures 22
2.2.2 Task allocation: related work 23
2.2.3 Schedulability analysis 23
2.3 Resource Sharing 25
2.3.1 Multiprocessor priority ceiling protocol 26
2.3.2 Multiprocessor stack resource policy 28
2.3.3 Wait-free methods 30
2.4 Mixed Criticality Systems 31
2.4.1 Basic MCS model 32
2.4.2 Schedulability analysis 34
2.4.3 MCS on multicores: related work 35
2.5 Fault-Tolerance in MCS 37
2.5.1 Lockstep execution and on-demand redundancy 38
2.5.2 Transient faults in MCS: related work 40
2.5.3 Permanent faults in MCS: related work 42
2.6 Model-Based Design Using Synchronous Reactive Models 43
2.6.1 SR semantics preservation 45
2.6.2 Model-based design: related work 50

3 Task Allocation in Multicore Real-Time Systems 52

3.1 System Model 54
3.2 Resource-Aware Task Allocation 55
3.3 Problem Formulation with MILP 56
3.3.1 General system variables and constraints 56
3.3.2 MSRP response time formulation 58

3.3.3 MPCP response time formulation 65
3.3.4 Objective function 73
3.4 Heuristic Algorithms 73
3.4.1 Extending greedy slacker with wait-free methods 74
3.4.2 Memory-aware partitioning algorithm 74
3.5 Experimental Results 80
3.5.1 Schedulability analysis 82
3.5.2 General evaluation of heuristics 84
3.5.3 Effect of different parameters 87
3.5.4 Evaluation with ILP 93
3.6 Conclusion 95

4 Task Allocation in Mixed-Criticality Systems 96
4.1 System Model 97
4.1.1 Elastic Mixed-Criticality (EMC) 97
4.1.2 Task model 99
4.1.3 System behavior 100
4.2 Dual-Partitioned Mixed-Criticality Scheduling 101
4.2.1 Schedulability Analysis 107
4.3 Experimental Evaluation 108
4.4 Conclusion 114

5 Tolerating Hardware Faults in Mixed-Criticality Systems 116
5.1 Scheduling MCS with Transient Faults: Motivation 118
5.2 Assumptions and Notation 120

5.2.1 Task model 120
5.2.2 Failure probabilities for transient faults 121
5.3 The Four-Mode System Model 121
5.3.1 Re-execution requirements in lockstep cores 123
5.3.2 System operation 124
5.3.3 Providing QoS to LO-criticality tasks 125
5.3.4 An illustrative example 125
5.4 Schedulability Analysis with Transient Faults 127
5.4.1 Reducing model pessimism 131
5.5 Generalization to ODR 132
5.6 DSE with Transient Faults 136
5.7 Experimental Results: Transient Faults 139
5.7.1 Experiments on single core architectures 140
5.7.2 Experiments on multicore architectures 142
5.8 Scheduling MCS with Permanent Faults: Motivation 145
5.9 System Operation with Permanent Faults 146
5.10 Schedulability Analysis with Permanent Faults 148
5.11 MILP-Based DSE with Permanent Faults 152
5.11.1 General system variables and constrains 152
5.11.2 Schedulability constrains 154
5.12 Experimental Results: Permanent Faults 157
5.13 Conclusion 160

6 Implementing Synchronous Reactive Models of Mixed-Criticality Systems 162

6.1 Schedulability vs Functional Delay Tradeoff: Example 164
6.2 System Model 166
6.2.1 Task model 166
6.2.2 EMC model with no mode change task dropping 167
6.2.3 System operation 168
6.3 Schedulability Analysis 170
6.4 Priority Assignment with Audsley's Algorithm 174
6.5 Optimization Algorithms 175
6.5.1 The Branch-and-Bound algorithm 175
6.5.2 The heuristic algorithm 183
6.6 Experimental Results 189
6.7 Conclusions 196

7 Conclusion and Future Work 198

7.1 Key Contributions 198
7.1.1 Resource-sharing multicores 198
7.1.2 Multicore mixed-criticality systems 199
7.1.3 Fault-tolerant mixed-criticality systems 200
7.1.4 Implementing synchronous reactive models of mixed-criticality systems 201
7.2 Summary of Best Practices 201
7.3 Future Work 202
7.3.1 Resource-sharing multicore mixed-criticality systems . . 202
7.3.2 Platform variations 203
7.3.3 Considering multiple viewpoints 203

7.3.4 Considering and quantifying migration overheads 203

7.3.5 MCS under multiple processor failure 204

7.3.6 A holistic fault-tolerant MCS model 204

List of Tables

2.1 Max errors for delays on the sensor and actuator paths 49

3.1 Task parameters for the ILP example system 61
3.2 Remote priority ceiling calculation 68
3.3 Schedulability/average memory cost (GS-WF and MPA only, in bytes) 83
3.4 Average runtime (in seconds) 84
3.5 MPCP schedulability/average memory cost (GS-WF and MPA only, in bytes) for different sharing factors 92
3.6 MSRP schedulability/average memory cost (GS-WF and MPA only, in bytes) for different sharing factors 93
3.7 MPCP Schedulability/average memory cost (in bytes) at different utilizations 94
3.8 MSRP Schedulability/average memory cost (in bytes) at different utilizations 94

4.1 Task Parameters for The Example System 104

5.1 An Example Task Set 126

5.2 An Example Task Set 131

5.3 Transformation parameters for the four fault-tolerance mechanisms. 134

5.4 Task set transformation 135

5.5 Rules for generating unique MS configurations from an integer x for n cores 138

5.6 An Example Task Set 148

6.1 Timing parameters for the example in Figure 6.2. 169

6.2 Notations used in this chapter 170

6.3 Timing parameters for the example in Figure 6.5. 186

List of Figures

1.1 The main componenets in an embedded systems design flow from a timing viewpoint 3

2.1 Basic job parameters 19
2.2 Classification of scheduling algorithms 21
2.3 Writer and readers stages in wait-free methods. 30
2.4 Architectures for multicore fault-tolerant systems. 39
2.5 Supported fault-tolerance mechanisms 40
2.6 Input/output relation with no edge delay. 47
2.7 Input/output relation with unit edge delay. 47
2.8 A Simulink example of an hydraulic servomechanism (representative of a suspension control). 48
2.9 Hydraulic servo with additional communication delays. 48
2.10 Actuator position and error for the hydraulic servo without (top) and with (bottom) delays. 49

3.1 MPCP remote ceiling example 68
3.2 Comparison of schedulability (8 cores, 4 resources) with MPCP 85

3.3 Comparison of schedulability (8 cores, 4 resources) with MSRP . 85
3.4 Comparison of memory cost (8 cores, 4 resources) with MPCP . 86
3.5 Comparison of memory cost (8 cores, 4 resources) with MSRP . 86
3.6 Comparison of schedulability (4 cores, 20 resources)- MPCP . . 89
3.7 Comparison of schedulability (4 cores, 20 resources) - MSRP . . 89
3.8 Comparison of memory cost (4 cores, 20 resources) - MPCP . . 90
3.9 Comparison of memory cost (4 cores, 20 resources) - MSRP . . 90
3.10 Comparison of schedulability with a variable number of resources - MPCP 91
3.11 Comparison of schedulability with a variable number of resources - MSRP 91

4.1 Example system execution trace 104
4.2 Percentage of schedulable systems at different LO-utilizations for various heuristics (before applying DPM) 110
4.3 Improvement on schedulability at different LO-utilizations . . . 111
4.4 Improvement on schedulability at different tasks counts 111
4.5 Improvement on schedulability at different percentages of HI tasks 112
4.6 Scalability of DPM to larger numbers of processors 112
4.7 Improvement at different values of the maximum CFactor 113
4.8 Improvement at different values of the maximum T(HI)/T(LO) ratio 114
4.9 Impact of Phase 2 114

5.1 The four-mode system model 122

5.2 An execution trace for the task set in Table 5.1 when (a) an overrun occurs; (b) a fault occurs; (c) both occur. 126
5.3 DSE workflow using nested genetic algorithms. 137
5.4 Modes OV and TF achieve better QoS than HI for all utilizations (F not bounded). 141
5.5 Modes OV and TF achieve better QoS than HI for different percentages of HI tasks (F not bounded). 141
5.6 Average improvement over all system utilizations for OV and TF modes compared to HI mode. 142
5.7 Performance of TF mode for different F. 142
5.8 ODR provides better QoS in multicore systems as utilization increases. 144
5.9 ODR provides better QoS in multicore systems as the percentage of HI tasks increases. 144
5.10 Combining ODR techniques improves QoS. 145
5.11 Combining ODR techniques improves schedulability. 145
5.12 An example system to illustrate the different scenarios to be considered in the schedulability analysis with failures. 149
5.13 Schedulability at different utilizations (N=20, L=4). 159
5.14 Schedulability at different utilizations (N=12, L=3). 159
5.15 Schedulability at different task set sizes ($U(LO)$=0.7, L=4): proposed (top), 2-step baseline (bottom). 160

6.1 Multirate SR models without edge delay, shown in (a), and with unit edge delay, shown in (b), and the corresponding schedules, shown in (c) and (d). 164
6.2 An example mixed-criticality SR model. The number in parentheses above a block denotes its period, and the number along the edge denotes its cost. 169
6.3 An example of EMC. 169
6.4 Example of a Branch-and-Bound search tree. 180
6.5 An example mixed-criticality synchronous program. The number in parentheses above a block denotes its period. 186
6.6 Schedulability of different algorithms vs. U_{LO} for AMC. 189
6.7 Schedulability of different algorithms vs. U_{LO} for EMC. 190
6.8 Schedulability of EMC vs. minimum period scaling factor $k_{\min}$, for different U_{LO}. 191
6.9 System cost vs. U_{LO} for AMC. 192
6.10 System cost vs. U_{LO} for EMC. 193
6.11 System cost vs. taskset size for AMC. 194
6.12 System cost vs. taskset size for EMC. 195
6.13 System cost and runtime of the heuristic for larger systems. . . 196

Chapter 1

Introduction

Embedded systems have become prevalent in all aspects of our lives today. Approximately 98% of the microprocessors sold today are used in embedded systems [1]. Embedded systems are already present in cars, airplanes, implantable medical devices, TVs, etc. They are also central to many emerging technologies such as wearable devices and the Internet of Things. Many of these systems are real-time embedded systems that have to interact with the environment within predefined timing constraints. "A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced" [2].

When designing such systems, the designer must make sure that the system always respects its timing constraints even for a worst case combination of internal and external events. This differs from traditional computing where emphasis has usually been on improving average case performance. In many real-time applications, a reaction that occurs too late could be useless or even

dangerous [3]. The emphasis on providing enough resources for the worst case behavior often has to be balanced with efficiency and other design constraints. Managing this tradeoff is not a trivial task, and designers need to handle often conflicting objectives of safety, efficiency, cost, and time-to-market.

In this dissertation, we focus on the design process of embedded real-time systems and propose new system models, analysis methods, and scheduling approaches. Our objective is to enable the design of efficient real-time systems that maximize the usage of the available hardware resources without compromising other system requirements such as timing and reliability constraints.

1.1 Timing in Embedded System Design

Figure 1.1 shows the main components of an example embedded design process from a timing viewpoint. The starting point for any design is a high level specification given as one or more documents describing the intended functionality of the system and the design requirements. Designers then convert this high level specification to a more refined implementation. This implementation needs to be deployed on a hardware platform. We focus on single core and homogeneous multicore hardware platforms where the implementation consists of one or more software programs or *tasks* that can execute on any core.

When multiple tasks run on a processor, the processor time allocated to each task must be determined. On a multicore platform, an additional complication is determining a suitable allocation of tasks to processors. This step is known as *task mapping* (also known as *task allocation* or *partitioning*). The set of rules that at any time, determines the order in which tasks are executed

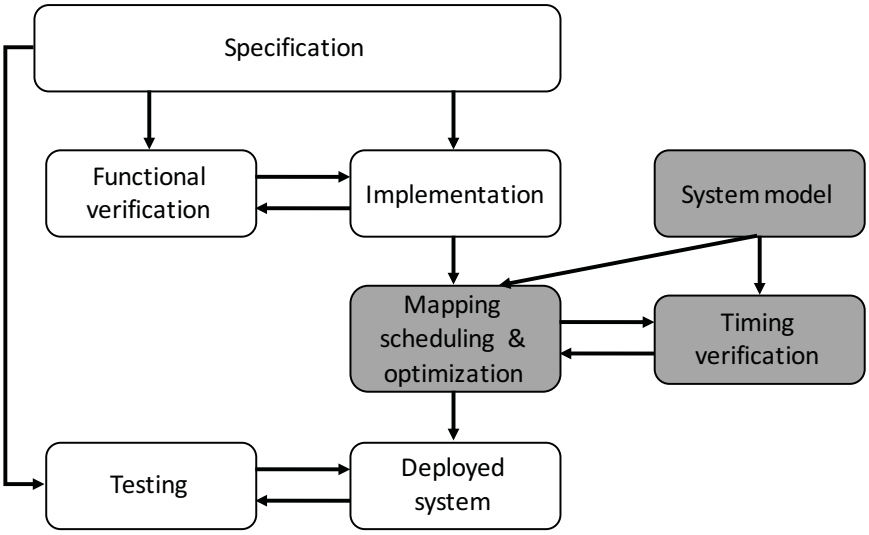


Fig. 1.1: The main componenets in an embedded systems design flow from a timing viewpoint

on a particular processor is called a *scheduling algorithm* [3].

This mapping and scheduling operation therefore takes an implementation consisting of a set of software tasks and a particular hardware platform and produces a deployed system. This is typically an iterative procedure where mapping and scheduling decisions are made and then revisited again and again to optimize one or more design objectives.

Before a fully deployed system is produced, the system must undergo comprehensive verification to ensure that the system's implementation matches the specification and requirements. For real-time systems, we need to both verify that the system produces the correct outputs (functional verification) and that it produces these outputs on time without violating the system's timing constraints (timing verification). Finally, the product goes through final testing.

Timing verification is typically done through formal analysis, often referred to as *schedulability analysis*, on abstract models of the system. This abstract system model focuses on those aspects that impact the timing verification process such as the number of processors, the number of tasks and their timing parameters. The system model also specifies other system features that can affect the timing analysis, for example: are tasks independent, or do they share resources? Do tasks have fixed priorities, or does a task's priority change over time? Are faults modeled, and if so, what is their impact on tasks? We will describe a basic timing model in Chapter 2 and adapt it in each of the remaining chapters, abstracting away details not relevant to the problem described in the chapter and adding new details.

In this work, we focus on the three "real-time related" components in Figure 1.1: a) the system model, b) timing verification, and c) mapping, scheduling and optimization. We propose new system models, associated analysis methods for verifying timing constraints, and new approaches for mapping and scheduling real-time systems.

1.2 Motivation

Designing real-time embedded systems is becoming an increasingly difficult task as consumers demand more features leading to more complex designs than ever before. This complexity accompanied with the pressure from increased competition is causing a reduction in the quality of the final products released to the market. Companies are often being forced to recall products leading to billions of dollars of losses, and sometimes unfortunately, it leads to tragic loss

of lives. The automotive industry, for example, where cost is a very important design constraint, is one of the most prominent examples of these problems. According to the National Highway Traffic Safety Administration [4], in 2013 the auto industry recalled 22 million cars, more than the number of cars sold that year [5]. This figure went up to 51 million in 2015 [6].

As the number of lines of code in a modern vehicle is rapidly increasing, software errors, whether due to functionally wrong code or timing errors, will constitute a significant part of design problems. This is also true for other industries such as avionics, medical devices, and consumer electronics. Our focus here is on timing-related errors. Many real-time system models and analysis methods have been proposed over the years. [7] provides a survey of some of the prominent real-time models for single core platforms. Tools such as Symtavision [8] and aiT [9] have been developed allowing designers to perform timing analysis of real-time systems.

However, there is always a need to adapt real-time models and propose new ones to accommodate emerging design and architecture trends such as the use of multicore architectures in real-time systems [10], and the integration of tasks with different criticalities on a common platform [11].

We will give special attention in this thesis to multicore architectures. Multicore architectures are becoming more common in real-time systems. The computational demands of application are increasing at a rapid pace. To address these demands, chip manufacturers have focused previously on increasing clock speeds. However, due to heating issues, this approach has become problematic. Instead, there is now an increasing trend towards using multiprocessor

platforms for high-end real-time applications [10].

Today's embedded systems are also more integrated than ever before. Another increasingly important trend in real-time systems is the integration of components of different levels of criticality onto a common hardware platform [12]. This led to the emergence of the concept of *Mixed-Criticality Systems* (MCS) [12]. The critical part of the system is often subject to certification requirements from certification authorities while the less critical tasks are not. Current certification standards, such as ISO26262 in the automotive domain and DO-178C in the avionics domain, define several criticality levels according to the level of assurance required for a given function.

Focusing on these design trends, we observe the following limitations in previous real-time models and scheduling techniques:

- *Mapping and scheduling in resource-sharing multicore systems*: tasks in real embedded systems often share resources such as shared variables, data structures, I/O resources, and peripherals. When real-time tasks allocated on different cores cooperate through the use of shared communication resources, they need to be protected by mechanisms that guarantee access in a mutually exclusive way with bounded worst-case blocking time. While exclusive access guarantees data consistency for the shared resource, it has a direct impact on scheduling and task allocation because of blocking delays. The problem of finding a feasible task allocation is known to be NP-hard (even in the case of no shared resources) [13]. Most works on task allocation and scheduling in multicore ignored the effect of shared resources. However, resource-agnostic task

allocation algorithms can introduce bottlenecks in the system through unnecessary distribution of tasks that share global resources across different processors [14].

- *Task mapping in multicore mixed-criticality systems*: MCS bring new and interesting challenges to the mapping problem. MCS can have multiple operational modes and inherently provide different guarantees to tasks in these modes. This complicates the mapping and scheduling problem as tasks can have different execution times in different modes. Traditional mapping and scheduling approaches do not extend to or become inefficient when applied to mixed-criticality systems.

- *Fault-tolerance in mixed-criticality systems*: MCS contain a safety critical part. That part of the system must be made reliable and should be able to survive transient or permanent faults occurring in the system. The problem of designing fault-tolerant MCS, however, has not yet received much attention [12]. Faults are typically tolerated using temporal and spatial redundancy. This redundancy imposes additional constraints on the design of the mixed-criticality system. System models, analysis techniques, and mapping and scheduling approaches for MCS should be adapted to take into account tasks' reliability requirements.

- *Model-based design for mixed-criticality systems*: model-based design is among the best practices for software development, especially in the automotive and aeronautic industry [15]. In a model-based design process, software implementations can be automatically generated from high

level models, reducing the number of errors injected by the design team. When a system implementation is generated automatically, the implementation might not be schedulable unless the designer modifies the model by adding functional delays. These delays have a negative impact on the system's cost and performance and should be avoided when possible. While there has been substantial focus on developing expressive functional models and languages, this has not been matched by a similar effort on the selection of the best mapping and the generation of optimized implementations. This is especially true in the case of mixed-criticality systems where efficient solutions to the problem of implementing a system developed using a model-based design process are still lacking

1.3 Thesis Statement

This thesis studies emerging trends in real-time system design. State-of-the-art real-time systems are composed of more functions that have different criticalities, share more resources, and are executing on more cores than ever before. With this observation in mind, we propose new mapping and scheduling approaches for resource-sharing multicore real-time systems. For mixed-criticality systems, we propose new system models and analysis methods for fault-tolerant MCS. We also propose new mapping and scheduling approaches for MCS that take into account the redundancy required for fault-tolerance. Moreover, we investigate generating MCS implementations in a model-based design flow. A persistent theme throughout the thesis is efficiency, i.e. pro-

ducing solutions that preserve design requirements while maximizing the use of available resources and/or minimizing implementation cost.

1.4 Thesis Contributions

To address the limitations in real-time models and design techniques discussed in Section 1.2, we make the following contributions in this thesis:

1.4.1 Mapping and scheduling in resource-sharing multicore systems

To overcome problems with resource-agnostic task allocation, we propose two novel resource-aware task allocation approaches for multicore real-time systems. Shared resources in a multicore system can be protected by lock-based mechanisms such as the Multiprocessor Priority Ceiling Protocol (MPCP) [16] and the Multiprocessor Stack Resource Policy (MSRP) [17], or by wait-free resources. Lock-based mechanisms have a negative impact on the schedulability of the system by forcing tasks to incur blocking while they wait for the resource to be unlocked. Wait-free methods can improve schedulability at the expense of additional memory. We exploit the tradeoff between lock-based and waitfree mechanisms and propose to consider the selection of the protection mechanism as an additional design variable. A more general design problem than previously explored is formulated through jointly performing task allocation, scheduling, and resource protection mechanism selection. The problem is formulated as a Mixed Integer Linear Program (MILP) for both MPCP and MSRP to find an optimal solution. Moreover, a heuristic algo-

rithm is proposed to solve the problem more quickly with good sub-optimal solutions for large systems where an MILP solution might take a long time to find. The solutions proposed produce more efficient task configurations than previously proposed algorithms, significantly extending the range of systems that can be scheduled on a particular platform. Experimental results indicate that state-of-the-art algorithms deteriorate quickly for systems beyond 65% utilization capacity and almost cease to schedule any systems beyond 80% utilization. Our proposed heuristic continues to schedule almost all systems at 80% utilization. The MILP achieves even better results. Section 3.5 provides a detailed comparison.

1.4.2 Task mapping in multicore mixed-criticality systems

As mentioned in Section 1.2, traditional task allocation algorithms can become inefficient in a mixed-criticality context. To preserve the guarantees provided to the critical tasks while efficiently executing non-critical tasks in the systems, we strike a balance between global and partitioned scheduling approaches and propose a novel semi-partitioned scheduling approach tailored for MCS. To this end, we propose the dual-partitioned mixed-criticality task allocation algorithm. Our experiments (Section 4.3) indicate that the dual-partitioned algorithm can schedule 17% more systems than traditional task allocation algorithms at system utilizations of 80% or higher.

1.4.3 Fault-tolerance in mixed-criticality systems

To incorporate concerns about reliability under hardware transient faults into MCS design, we propose a new MCS system model that jointly addresses both reliability and criticality requirements. In the new model, critical tasks are guaranteed sufficient service under transient faults to satisfy reliability requirement from standards such as DO-178C [18]. We derive schedulability analysis for the new model. The model and analysis support various fault-tolerance mechanisms such as re-execution, Dual Modular Redundancy (DMR), Triple Modular Redundancy (TMR), and passive replication. Moreover, we present a design space exploration approach that explores different task mappings and fault-tolerance mechanisms. The proposed exploration approach aims to find designs that satisfy criticality and reliability requirements while also addressing efficiency by maximizing the service provided to non-critical tasks.

We also consider permanent hardware faults in MCS and present a design approach for multicore MCS that can survive a processor failure. To this end, we extend the system operation description and schedulability analysis of the standard MCS model to incorporate the failure of one of the processors. An MILP-based mapping and scheduling approach is proposed to produce MCS designs that are efficient, tolerant to permanent faults, and adhere to criticality requirements. The possibility of processor failures is taken pro-actively into account when making the initial mapping and scheduling decisions. We show that this design approach improves schedulability significantly (for example a 3.2X improvement is observed for systems composed of 20 tasks and 4 cores)

over a design approach that adds reliability on top of an initial reliability-agnostic system design.

1.4.4 Implementing synchronous reactive models of mixed-criticality systems

Various models of computation have been used in model-based design for embedded systems. We focus here on the Synchronous Reactive (SR) model of computation which is used extensively for control applications and is the underlying model of computation for many formalisms and languages such as Simulink, Esterel, LUSTRE, and SIGNAL. We propose efficient solutions to the problem of implementing synchronous reactive models of MCS. Our objective is to ensure the preservation of the model semantics in the implementation while minimizing the use of functional delays. An optimal branch-and-bound based algorithm and a heuristic algorithm are proposed for this purpose. Our experimental evaluation in Section 6.6 shows that the heuristic algorithm produces close to optimal results (only about 3% higher cost) while being scalable to large systems.

1.5 Related Publications

The work in this thesis resulted in several publications which are listed below.

1. [19] Z. Al-bayati, Y. Sun, H. Zeng, M. Di Natale, Q. Zhu and B. H. Meyer, "Task placement and selection of data consistency mechanisms

for real-time multicore applications," 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15), pp. 172-181.

In this paper, we propose a resource-aware task allocation algorithm for systems that use MSRP to protect shared resources. Furthermore, we leverage the additional opportunity provided by wait-free methods and propose an algorithm that performs both task allocation and data consistency mechanism selection (MSRP or wait-free). Results show that the selective use of wait-free methods can significantly extend the range of schedulable systems at the cost of memory. This paper is covered in Chapter 3.

2. [20] Z. Al-bayati, Q. Zhao, A. Youssef, H. Zeng and Z. Gu, "Enhanced partitioned scheduling of Mixed-Criticality Systems on multicore platforms," The 20th Asia and South Pacific Design Automation Conference (ASPDAC'15), 2015, pp. 630-635.

   In this paper, the efficient partitioning of MCS on multicore architectures is discussed. A novel mixed-criticality partitioning algorithm, the Dual-Partitioned Mixed-Criticality (DPM) algorithm, is presented. Experimental results show that DPM consistently outperforms existing mixed-criticality partitioning algorithms, for example, at utilizations of 0.8 or higher, DPM is able to schedule 17% more systems. This paper is discussed in Chapter 4.

3. [21] Z. Al-bayati, J. Caplan, B. H. Meyer and H. Zeng, "A four-mode model for efficient fault-tolerant mixed-criticality systems," 2016 Design,

Automation and Test in Europe Conference & Exhibition (DATE'16), 2016, pp. 97-102.

In this paper, we consider the problem of designing and scheduling certifiable fault-tolerant mixed-criticality systems under transient faults. We propose a new four-mode model that addresses faults and execution time overruns with separate modes. This model, combined with the selective continuation of low-criticality tasks, improves the Quality of Service (QoS) to these tasks while providing the same guarantee to high-criticality tasks. Experimental results show that QoS improvements of up to 42.9% can be achieved by the new model. This paper is discussed in Chapter 5.

4. [22] Z. Al-bayati, B. H. Meyer and H. Zeng, "Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures," 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'16), 2016, pp. 57-62.

   In this paper, we present an approach to design multicore mixed-criticality systems that can survive permanent processor failures. Critical tasks executing on the failing cores are migrated to other cores to allow them to continue execution. Schedulability analysis for the extended model is developed. Then, the problem of finding a mixed-criticality system configuration on a multicore architecture is formulated as a MILP. This paper is discussed in Chapter 5.

5. [23] Q. Zhao, Z. Al-bayati[1], H. Zeng, and Z. Gu, "Optimized Implementation of Multi-Rate Mixed-Criticality Synchronous Reactive Models," ACM Transactions on Design Automation of Electronic Systems (TODAES). Volume 22 Issue 2, Article 23. January 2017.

   The paper considers model-based design using SR models for MCS. The paper presents a branch-and-bound procedure and a heuristic algorithm to minimize the total system cost of functional delays in the implementation. This paper is discussed in Chapter 6.

   Two other journal papers are still under review:

6. Z. Al-bayati, Y. Sun, H. Zeng, M. Di Natale, Q. Zhu and B. H. Meyer, "Partitioning and Selection of Data Consistency Mechanisms for Multicore Real-Time Systems".

   This paper extends [19] with support for the MPCP suspension-based protocol. An MILP-based solution method is also presented to perform task allocation and data consistency mechanism selection in an optimal way for both MPCP and MSRP. The paper is discussed in Chapter 3.

7. J. Caplan, Z. Al-bayati[2], H. Zeng, and B. H. Meyer, "Mapping and Scheduling Mixed-Criticality Systems with On-Demand Redundancy".

   This paper extends the model and analysis in [21] to support on-demand redundancy with various fault tolerance mechanisms. A design space

[1] My contribution for this paper was being the main developer of the branch-and-bound based algorithm and phase 2 of the heuristic with input from Ms. Zhao. I also implemented the parts I contributed.

[2] For this paper, the core model was co-developed with Mr. Caplan. I also contributed the derivation of the generic re-execution profiles.

exploration approach based on the improved analysis is also presented. The paper is discussed in Chapter 5.

1.6 Thesis Outline

Chapter 2 presents background information essential to the understanding of the rest of the thesis and discusses related works.

Chapter 3 presents our resource-aware mapping and scheduling approaches for mulitcore real-time systems. An optimal MILP-based solution is presented followed by a sub-optimal heuristic. Experimental results comparing these approaches to state-of-the-art algorithms are also presented.

Chapter 4 presents a novel efficient task allocation approach specific for MCS. Experimental results are presented comparing this approach with traditional task allocation approaches.

Chapter 5 presents our solutions for designing fault-tolerant MCS under both transient and permanent faults. For both types of faults, an MCS model, its analysis, and accompanying design space exploration approaches are presented.

Chapter 6 presents two algorithms for implementing MCS designed using a model-based design process. Experiments exploring the performance and scalability of both algorithms are presented.

Chapter 7 concludes the thesis highlighting the main findings and outlining areas for future improvement.

Chapter 2

Background and Related Work

In this chapter, we present background information on system models and analysis in real-time systems focusing especially on resource-sharing multicore systems and Mixed-Criticality Systems (MCS). We also discuss relevant related works. The chapter starts by presenting a basic model for real-time systems in Section 2.1. In Section 2.2, we discuss scheduling and task allocation. In Section 2.3, we discuss the effect of introducing shared resources into the scheduling problem. In Section 2.4, we give a brief introduction to MCS. In Section 2.5, we discuss fault-tolerance in the context of MCS. Finally, in Section 2.6, we give a brief overview of model-based design.

2.1 System Model: Basic Definitions

In this section, we describe a generic (from the timing viewpoint) system model. We consider a system in which the functionality can be divided into a set of schedulable units called *tasks*. A system consists of a set of N tasks

$\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$, and M homogeneous processors $P = \{p_1, p_2, ..., p_M\}$. For single core architectures, M=1. A task is the smallest entity scheduled by the operating systems. Each instance of a task is called a *job*. Tasks can generate an infinite sequence of jobs. We denote the jth instance (job) of a task τ_i by $\tau_{i,j}$. A job $\tau_{i,j}$ can be characterized by the following parameters (summarized in Figure 2.1):

- Arrival time (a_i^j): the time instant at which the job becomes ready to execute.
- Finish time (f_i^j): the time instant at which the job finishes its execution.
- Execution time (C_i^j): the total amount of time that the job spends executing on the processor (sum of the two shaded regions in Figure 2.1).
- Interference (I_i^j): the total amount of time that the job spends waiting for other jobs to finish their execution (sum of I_i^{j1} and I_i^{j2} in Figure 2.1).
- Relative deadline (D_i^j): the amount of time that the task is given to complete after its arrival. This is typically the same for all jobs of task (i.e. $\forall$ jobs j, k of task τ_i: $D_i^j = D_i^k = D_i$).
- Response time (R_i^j): the total amount of time the job takes from arrival until it finishes execution including interference from other tasks . $R_i^j = f_i^j - a_i^j$
- Inter-arrival time or period (T_i^j): the amount of time between the arrival of the job and the arrival of the subsequent job of the task $T_i^j = a_i^{j+1} - a_i^j$.

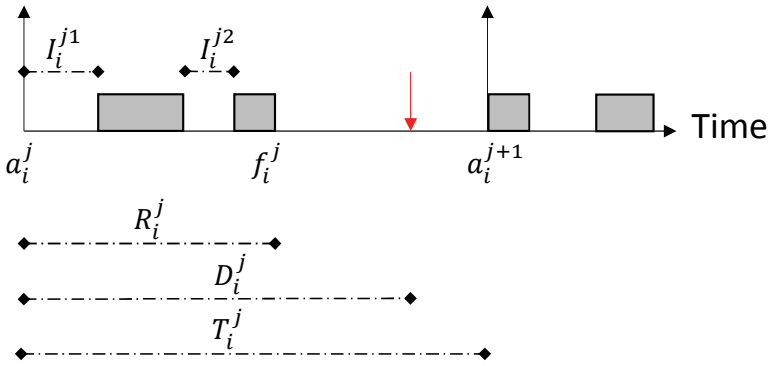


Fig. 2.1: Basic job parameters

Tasks that release jobs in an irregular manner (arrival times of the task's jobs are unrelated) are called *aperodic tasks*. Tasks that release jobs at regular intervals are called *periodic tasks*. For periodic tasks, the amount of time between the arrivals of two subsequent jobs is constant. Between the two types is another type of tasks that releases jobs with a minimum inter-arrival time. After this time passes, a new job can arrive at any instant. These tasks are called *sporadic tasks*. In this work, we are concerned with the more common periodic and sporadic task sets.

In real-time scheduling, we are interested in the worst case behavior of systems. Therefore, we characterize a task by deriving the worst case parameters across all job instances. **A task τ_i in the simplest case can be fully characterized by the tuple** $\langle C_i, D_i, T_i \rangle$ where

- C_i: the Worst Case Execution Time (WCET) of the task given by $C_i = \max_j(C_i^j)$. The WCET of the task is usually derived using static code analysis tools or measured by simulation.

- T_i: the minimum inter-arrival time of the task $T_i = \min_j(a_i^{j+1} - a_i^j)$.
- D_i: the task's relative deadline. This is typically dependent on the application requirements.

The deadline of the task could be smaller, equal to, or greater than its period. To simplify the discussion we will restrict our scheduling discussion to implicit deadline systems ($D = T$) and constrained deadline systems ($D \leq T$). The ratio C_i/T_i is referred to as the task utilization u_i.

2.2 Scheduling

Scheduling is one of the key issues in real-time systems design. The scheduling problem consists of determining which tasks execute on the processors at each time instant. The scheduling problem in the general sense is known to be NP-complete [24]. The different properties of real-time designs and the different assumptions used by systems' developers led to the development of a large number of scheduling algorithms. There are different criteria to classify scheduling algorithms. Figure 2.2 [25] shows one possible classification.

As mentioned in Section 2.1, we are interested here in periodic and sporadic hard real-time systems that are common in many applications and domains today such as control, automotives, and avionics. A *preemptive* scheduling algorithm can suspend a running job to allow a high priority job to run. We will focus here on preemptive algorithms since they allow higher efficiency [3]. A static scheduling algorithm makes scheduling decisions at deign time, for example the scheduler is given a table of task start and and end times which

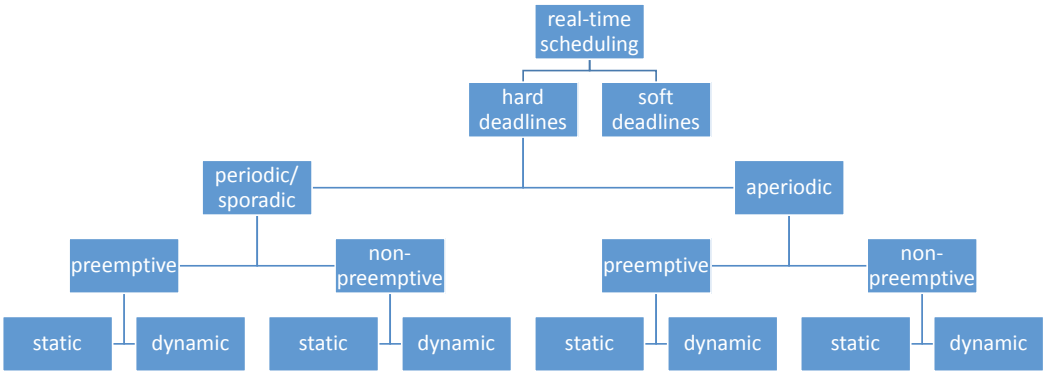


Fig. 2.2: Classification of scheduling algorithms

the scheduler enforces. Dynamic scheduling algorithms make their decisions at runtime. Static scheduling is efficient, however, it lacks the flexibility to deal with possible runtime scenarios. Furthermore, generating the schedule tables might be difficult for large systems with non-harmonic parameters, and these tables if found could be long and have high memory requirements. We will focus here on dynamic scheduling.

Dynamic scheduling of real-time systems is typically based on priorities assigned to tasks. The priority of the task can remain constant during the system's operation (*fixed priority scheduling*) or the priority of the task can change dynamically at runtime (*dynamic priority scheduling*). Each type of scheduling has its own advantages and disadvantages. Fixed priority scheduling algorithms such as the Rate Monotonic (RM) scheduling algorithm are simpler to implement and more predictable. Dynamic priority scheduling algorithms such as Earliest Deadline First (EDF) can potentially schedule more systems, however, they are hard to implement and in many instances are not supported by standards. [3] provides a more detailed comparison.

2.2.1 Scheduling in multicore architectures

When multicore architectures are the implementation target, another dimension to the scheduling problem is the selection of the processor on which a given job is executed. There are two general approaches:

- Global Scheduling: jobs are allowed to migrate freely between processors. A job can start execution on one processor and end its execution on another processor

- Partitioned Scheduling: Tasks are allocated to a certain processor. All jobs of the task must start and finish execution on this processor and no migration is allowed

Global scheduling has the potential to achieve better utilization for the overall systems. Partitioned scheduling can be implemented more easily and predictably, also it has the practical advantage that once a task allocation is found, a wealth of real-time scheduling techniques and analyses for uniprocessor systems can be used [10].

Therefore, in this thesis we focus on **partitioned fixed-priority scheduling** where tasks are statically assigned to cores and each core is scheduled by a local fixed-priority scheduler. This is widely used in embedded multicore real-time systems today. Such scheduling policies are supported by the AUTOSAR standard for automotive systems [26], as well as most commercial RTOSes, including VxWorks, QNX, LynxOS, and all POSIX-compliant ones.

For fixed-priority partitioned systems, we denote the priority of a task τ_i by π_i and the processor to which τ_i is assigned by P_i. Lower priority values

indicate a higher priority task i.e. if $\pi_i < \pi_j$ then τ_i has a higher priority. Designers of these systems must determine an appropriate allocation of tasks to processing cores as well as assign priorities on each core. At runtime, the highest priority task at each time instance executes on the processor. Designers need to ensure that all tasks finish their execution before their deadlines. This is done using schedulability analysis (discussed in Section 2.2.3).

2.2.2 Task allocation: related work

Allocation problems are very common in multicore systems, and in most cases they are proven to be special instances of the general bin-packing problem [27]. For real-time systems without considering shared resources, several heuristics are proposed, e.g., [13, 28, 29]. Baruah and Bini presented an exact ILP (Integer Linear Programming) based approach [30]. Chattopadhyay and Baruah showed how to leverage lookup tables to enable fast, yet arbitrarily accurate partitioning [31]. Baruah presented a polynomial-time approximation scheme [32]. Other results for partitioned scheduling of independent sporadic tasks can be found in a recent survey [10]. The work in Chapter 3 of this thesis differs from these works by considering the impact of shared resources when performing task allocation.

2.2.3 Schedulability analysis

Schedulability analysis is used to check that all tasks in the system meet their deadline requirements. For fixed priority systems, one of the most common methods to check whether a system is schedulable is response time anal-

ysis [3, 33]. For the model presented in this chapter, the worst case response time of a task τ_i, denoted by R_i occurs when the task is released simultaneously with all higher priority tasks and is given by:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{2.1}$$

where $hp(i)$ is the set of tasks that have higher priority than τ_i. Note that R_i appears on both sides on the equation. The response time is the smallest value that satisfies Equation (2.1). This value can be found iteratively as illustrated by the following equation:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \tag{2.2}$$

The iterations start by setting a suitable initial value for R_i^n (e.g. C_i since $R_i \geq C_i$). The right hand sides is then evaluated and the result is assigned to R_i^{n+1}. In the second iteration, this result is used on the right hand side as R_i^n and a new R_i^{n+1} is calculated. The process continues until no increase in R_i^{n+1} occurs for two consecutive iterations (i.e. R_i^{n+1}=R_i^n). This result represents the response time. The iterations also stop if R_i^{n+1} grows larger than the deadline D_i as this indicates that the task is not schedulable.

Once the response times for all tasks are calculated, a system is schedulable if and only if:

$$\forall \tau_i \in \Gamma : R_i \leq D_i \tag{2.3}$$

2.3 Resource Sharing

A resource is a software structure used by a task to advance its execution such as a variable, a data structure, or a set of registers on a peripheral device [3]. When a resource is shared between tasks, it might be possible that more than one task attempts to access the resource concurrently. This can be dangerous as these accesses might leave the shared resource in an inconsistent state. Protection mechanisms must be introduced to ensure that the resource remains in a consistent state. This is typically done through synchronization locks. Before getting access to the resource, a task must first attempt to lock the shared resource. If the locking operation is successful, the task is granted access to the resource. Once done, the task releases the lock. If the locking operation is not successful, the task must wait for the resource to be released and is said to be *blocked* on the resource. In a multicore architecture, the resource can be local (accessed by tasks allocated to the same processor) or global (accessed by tasks allocated to different processors).

The period of time a task τ_i spends accessing a shared resource is typically referred to as a *critical section*. The execution of task τ_i is composed of a set of alternating critical sections and sections in which τ_i executes without using a (global or local) shared resource, defined as *normal execution segments*. The WCET is defined by a tuple $\{C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,s(i)-1}, C_{i,s(i)}\}$, where $s(i)$ is the number of normal execution segments, and $s(i) - 1$ is the number of critical sections. $C_{i,j}$ ($C'_{i,j}$) is the WCET of the j-th normal execution segment

(critical section) of τ_i. The WCET C_i of τ_i is then

$$C_i = \sum_{1 \leq j \leq s(i)} C_{i,j} + \sum_{1 \leq j < s(i)-1} C'_{i,j} \tag{2.4}$$

The time a given task τ_i spends waiting for a shared resource to become free is commonly referred to as *blocking time,* B_i. If τ_i accesses more than one resource, then B_i is simply the sum of the blocking times encountered across all resources. To ensure resource accesses proceed with bounded blocking times and without deadlocks, resource access protocols have been proposed. These protocols manage access to the shared resources and determine task priorities during their critical sections. On multicore architectures, two prominent protocols have been used extensively in academia and industry: suspension-based Multiprocessor Priority Ceiling Protocol (MPCP) [16], and spin-based Multiprocessor Stack Resource Policy (MSRP) [17].

MPCP and MSRP have been compared in a number of research works (with respect to their worst-case timing guarantees), with a general consensus that MSRP performs best for short (global) critical sections and MPCP for long ones [34]. We will describe these two protocols next.

2.3.1 Multiprocessor priority ceiling protocol

MPCP [16] is a multiprocessor extension of the Priority Ceiling Protocol (PCP) [35]. In MPCP, tasks use assigned priorities for normal executions, and inherit the ceiling priority of the resource whenever they execute a critical section. The ceiling priority of a local resource is the highest priority of any task that can possibly use it. For a global resource, its remote priority ceiling is required to be higher than any task priority in the system. For this purpose,

a base priority offset higher than the priority of any task is applied to all global resources. Jobs are suspended when they try to access a locked global resource and added to a priority queue. The suspension of a higher priority task blocked on a global critical section allows other (possibly lower priority) tasks to execute and may even try to execute local or global critical sections. We briefly review the schedulability analysis of MPCP [14, 36].

The normal section of a task can be blocked by the critical section of lower priority tasks on the same core. For each of the $s(i)$ normal sections, the worst case local blocking time B_i^l that τ_i may suffer is the longest critical section among all the lower priority local tasks:

$$B_i^l = s(i) \times \sum_{k:\pi_k>\pi_i \& P_k=P_i} \max_{1\le m<s(k)} C'_{k,m}. \tag{2.5}$$

τ_i can only be interfered with by critical sections with a higher remote priority ceiling once it enters its critical section. Also, since the critical section has higher priority than the normal section, in a critical section τ_i will only suffer one such interference from each task on the same core. Thus, the response time of the j-th critical section is bounded by:

$$W'_{i,j} = C'_{i,j} + \sum_{k\neq i:P_k=P_i} \max_{1\le m<s(k)\&\Pi_{k,m}<\Pi_{i,j}} C'_{k,m}. \tag{2.6}$$

where $\Pi_{i,j}$ is the priority ceiling of the critical section $C'_{i,j}$. The remote blocking time $B_{i,j}^r$ suffered in the j-th critical section is:

$$\begin{aligned} B_{i,j}^r &= \max_{\pi_k>\pi_i \& \mathcal{S}_{k,m}=\mathcal{S}_{i,j}} W'_{k,m} \\ &+ \sum_{\pi_h<\pi_i \& \mathcal{S}_{h,n}=\mathcal{S}_{i,j}} \left(\left\lceil \frac{B_{i,j}^r}{T_h} \right\rceil + 1 \right) W'_{h,n}, \end{aligned} \tag{2.7}$$

where $\mathcal{S}_{i,j}$ denotes the resource accessed by the critical section $C'_{i,j}$ and $\mathcal{S}_{k,m} = \mathcal{S}_{i,j}$ indicates that the critical sections $C'_{i,j}$ and $C'_{k,m}$ access the same resource.

The initial value for the iterative procedure can be set as the first term on the right hand side. The total remote blocking time is then:

$$B_i^r = \sum_{1 \le j < s(i)} B_{i,j}^r. \tag{2.8}$$

In [14], the worst case response time R_i of τ_i is then calculated by the following iterative formula:

$$R_i = C_i + B_i^l + B_i^r + \sum_{\pi_h < \pi_i \& P_h = P_i} \left\lceil \frac{R_i + B_h^r}{T_h} \right\rceil C_h. \tag{2.9}$$

Recently, Chen et al. [36] showed that Eq. (2.9) is optimistic, and proposed a fix by replacing B_h^r with $R_h - C_h$ when $R_h \le T_h$. The response time calculation becomes:

$$R_i = C_i + B_i^l + B_i^r + \sum_{\pi_h < \pi_i \& P_h = P_i} \left\lceil \frac{R_i + R_h - C_h}{T_h} \right\rceil C_h. \tag{2.10}$$

2.3.2 Multiprocessor stack resource policy

MSRP (Multiprocessor Stack Resource Policy) [17] is a multiprocessor extension of the Stack Resource Policy (SRP) [37]. The local shared resources are managed in the same way as MPCP. The global resources are assigned with a ceiling that is higher than that of any local resource. A task that fails to lock a global resource *spins* on the resource lock until it is freed, keeping the processor busy. (In the MPCP protocol, for comparison, the task is *suspended* and yields the CPU.) To minimize the spin lock time (wasted CPU time), tasks cannot be preempted when executing a global critical section, so as to free the resource as soon as possible. MSRP uses a FIFO queue (as opposed to a priority-based queue in MPCP) to manage the tasks waiting on a lock for

a given busy resource. MSRP maintains the same basic property of SRP: once a task starts execution it cannot be blocked.

We briefly review the sufficient schedulability analysis for MSRP in [17]. Recently, a more accurate analysis based on an ILP formulation has been proposed by Wieder and Brandenburg [38]. However, for our objective of using the analysis in design optimization where many solutions need to be evaluated quickly, the significant runtime of this approach causes scalability issues in large systems. This will be demonstrated with experimental results in Chapter 3.

The local blocking time B_i^l of task τ_i is bounded by the longest critical section of a lower priority task on the same core accessing a resource with a ceiling higher than π_i.

$$B_i^l = \max_{k:\pi_k>\pi_i \wedge P_k=P_i} \left\{ \max_{1\leq m<s(k)} C'_{k,m} \right\} \tag{2.11}$$

The spin time $L_{i,j}$ that τ_i may spend for accessing a global resource $\mathcal{S}_{i,j}$ can be bounded by:

$$L_{i,j} = \sum_{P\neq P_i} \left\{ \max_{\tau_k:P_k=P, 1\leq m<s(k)} C'_{k,m} \right\}. \tag{2.12}$$

The total worst case execution time C_i^* becomes:

$$C_i^* = C_i + \sum_{1\leq j<s(i)} L_{i,j}. \tag{2.13}$$

The worst-case remote blocking time B_i^r is:

$$B_i^r = \max_{k:\pi_k>\pi_i \wedge P_k=P_i} \left\{ \max_{1\leq m<s(k)} (C'_{k,m} + L_{k,m}) \right\}. \tag{2.14}$$

Finally, the worst case response time R_i of τ_i is:

$$R_i = C_i^* + \max\{B_i^l, B_i^r\} + \sum_{\pi_h<\pi_i \wedge P_h=P_i} \left\lceil \frac{R_i}{T_h} \right\rceil C_h^*. \tag{2.15}$$

2.3.3 Wait-free methods

When the shared resource is a **communication buffer** (memory used for communicating data), lock-based methods can be replaced with **wait-free methods** for ensuring data consistency. The writer and readers are protected against concurrent access by replicating the communication buffer and leveraging information on the time instant and order (priority and scheduling) of the buffer access [39, 40]. Wait-free methods have virtually no blocking time (in reality often negligible) and may enhance the schedulability of the system at the cost of additional memory.

The objective of wait-free methods is to avoid blocking by ensuring that each time a writer needs to update the communication data, it is reserved with an unused buffer. Readers are free to use other dedicated buffers. Figure 2.3 shows the typical stages performed by the writer and the readers in a wait-free protocol implementation [41].

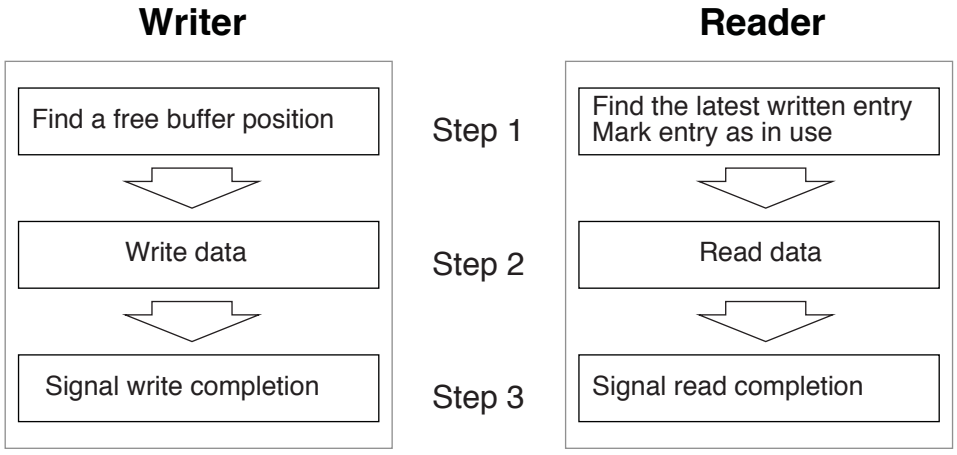


Fig. 2.3: Writer and readers stages in wait-free methods.

The algorithm makes use of three global sets of data. An array of buffers

is sized so that there is always an available buffer for the writer to write new data. An array keeps in the i-th position the buffer index in use by the i-th reader. Finally, a variable keeps track of the latest buffer entry that has been updated by the writer. Each reader looks for the latest entry updated by the writer, stores its index in a local variable, and then reads its contents.

Consistency in the assignment of the buffer indexes can be guaranteed by any type of hardware support for atomic operations, including Compare-And-Swap (CAS) (as in the original paper by Chen and Burns [41]), but also Test and Set, and hardware semaphores, as available in several architectures today (the required support for atomicity is discussed in detail in [42]). The use of hardware support for atomicity is not limited to wait-free methods. Any implementation of MSRP for example needs similar instructions for the low-level protection of the lock data structures (including the FIFO queue for tasks waiting for the shared resource).

2.4 Mixed Criticality Systems

Embedded systems today are performing more complex and diverse tasks than ever before, as cost constraints drive the integration of diverse features in highly integrated systems. Some of these features might be critical to the operation or the safety of the system, thus requiring explicit certification, while others do not. *Criticality* is the level of assurance needed by a particular task in the system [12]. Systems composed of components with different criticalities are called *Mixed Criticality Systems* (MCS). MCS have numerous applications in domains such as automotive and avionics. These systems are subject of

ongoing research summarized in [12]. Current certification standards such as ISO26262 in the automotive domain and DO-178C in the avionics domain define several criticality levels according to the level of assurance required for a given function.

MCS are often subject to certification by regulators. To ensure safety, certification authorities require that the correctness of the system be demonstrated under extremely conservative constraints that are unlikely to arise in reality [43]. This leads to higher estimates of tasks' execution times than those calculated by designers. To obtain certification, designers must able to demonstrate that the safety-critical part of the system meets its timing constraints even with these high estimates. MCS assume that the underlying operating system is trusted, certified to the highest criticality level, and provides monitoring and isolation services. This type of OS, with minor variations in features, can be called a microkernel, a hypervisor, or separation kernel [44].

2.4.1 Basic MCS model

The introduction of levels of criticality into the design of real-time systems requires revisiting the theory and models of real-time scheduling to take into account the new challenges introduced by MCS. *Mixed-criticality scheduling* [11] was proposed to achieve efficient utilization of the underlying CPU resources while guaranteeing safety and temporal isolation for highly critical tasks. Since this first paper, many other works have addressed mixed criticality scheduling. The work on MCS scheduling has resulted in a standard model [45, 46].

The standard model introduces a new task parameter, criticality level L. Two criticality levels for the systems' tasks are typically defined ($L \in \{LO, HI\}$): LO (for non-critical tasks) and HI (for critical tasks). Non-critical tasks have a single execution time ($C(LO)$). Safety critical tasks are characterized by two execution times ($C(LO)$, $C(HI)$) where $C(HI) \geq C(LO)$. These execution times represent the designer's and the certification authority's estimates respectively. For scheduling tasks according to the standard MCS model with fixed priority scheduling, a new scheduling scheme, called Adaptive Mixed-Criticality (AMC) [46] was introduced. Under AMC, system operation proceeds as follows [46]:

1. The system starts operating in the LO mode.

2. Under the LO mode, all systems tasks (both LO and HI) are allowed to execute. The task with the highest priority (regardless of criticality) at any time is assigned to the processor.

3. If any task in the system tries to execute beyond its LO-mode WCET ($C(LO)$) without signalling completion, the system switches into the HI mode.

4. When this happens, LO tasks are dropped immediately. The HI task with the highest priority executes on the processor. Under the HI mode, these HI tasks are allowed to execute upto their certification authority accepted WCET estimate ($C(HI)$).

The HI mode ensures that even if the conservative estimate from the regulator is accurate, the safety-critical part of the system will continue to operate.

The LO mode demonstrates normal operation of the full system with more optimistic WCET estimates.

2.4.2 Schedulability analysis

The schedulability of MCS executing according to the AMC model can be verified using response time analysis. A sufficient schedulability test known as AMC-rtb is presented in [46]. The work in [46] also demonstrates that an exact test would not be tractable. Under AMC-rtb, three cases need to be verified [46]:

1. the schedulability of all tasks in the LO mode;
2. the schedulability of HI tasks in the HI mode; and,
3. the schedulability of HI tasks during the LO to HI mode .

The response time $R_i(LO)$ of a task τ_i in the LO mode is:

$$R_i(LO) = C_i(LO) + \sum_{j \in hp(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil \cdot C_j(LO) \tag{2.16}$$

Once a mode change starts, the LO tasks are not allowed to execute any more; we are therefore only concerned with checking the schedulability of the HI tasks in the stable HI mode and during the mode change itself. The response times of a HI task in the HI mode and mode change are:

$$R_i(HI) = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil \cdot C_j(HI) \tag{2.17}$$

$$R_i(MC) = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i(MC)}{T_j} \right\rceil \cdot C_j(HI) + \sum_{k \in hpL(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil \cdot C_k(LO) \tag{2.18}$$

where $hpH(i)$ ($hpL(i)$) is the set of HI-criticality (LO-criticality) tasks that have the same processor as τ_i and have higher priority. To affect an instance of τ_i, a mode change must occur before $R_i(LO)$. Hence tasks in $hpL(i)$ are dropped after $R_i(LO)$ and we need to account for them only until $R_i(LO)$. Eq.(2.18) contains the same terms as Eq.(2.17) plus one additional term. Hence, Eq.(2.17) is subsumed by Eq.(2.18), and in practice we only need to verify Eq.(2.16) and Eq.(2.18).

2.4.3 MCS on multicores: related work

Scheduling MCS on multicores has received significant attention recently due to its relevance for current and future high performance MCS. One of the first works on multi-core scheduling of MCS is [47]. Five different criticality levels are employed with each level using a different form of scheduling and partitioning. Level-A (the highest level) uses a cyclic executive (static scheduling), level-B uses partitioned EDF, levels C and D use global EDF, and level-E uses global best effort. For scheduling tasks, a complex two-level hierarchical scheduler is used. The approach also placed some restriction on tasks' periods requiring all periods of level-B tasks be integer multiples of the level-A hyperperiod [47].

In [48], Baruah et al. compare the use of partitioning and global scheduling for MCS and conclude that partitioned scheduling is generally more effective.

A major reason is that the utilization bounds used as schedulability tests for global scheduling are often pessimistic whereas partitioned scheduling can use more accurate schedulability analysis techniques [48]. Moreover, partitioned scheduling is adopted in the industrial standards for multicore real time systems scheduling [49].

For systems scheduled with dynamic priorities (EDF), several recent works discuss partitioning for MCS scheduled by EDF-VD [50] (EDF with Virtual Deadlines, an extension of EDF for MCS), such as [49, 51]. In [51], a partitioning scheduling algorithm called MPVD is proposed. In MPVD, HI-criticality tasks are allocated using the Worst-Fit bin-packing algorithm then LO-criticality tasks are allocated using First-Fit. Tasks are sorted by their utilization at their criticality level in both cases. Rodriguez et al. [49] use a similar method trying more combinations of bin-packing algorithms (Best-Fit, Worst-Fit, First-Fit and Next-Fit). The authors conclude that a scheme that assigns HI tasks first using Worst-Fit then LO tasks using First-Fit both sorted by decreasing density maximizes the schedulability success ratio.

For fixed priority scheduling, [52] compares various bin-packing heuristics for partitioned scheduling of MCS. Tasks are ordered by decreasing utilization or by decreasing criticality and the partitioning algorithms First-Fit, Best-Fit, and Worst-Fit are compared. First-Fit and Best-Fit with decreasing criticality are found to have the best success ratios. It is worth noting that the paper uses the older Static Mixed Criticality (SMC) [11] model which has largely been superseded by the more recent AMC [46].

In Chapter 4, a hybrid scheduling approach (between global and parti-

tioned scheduling) for multicore MCS is proposed. The proposed approach tries to combine the advantages of both scheduling appoaches while taking into account the nature of MCS.

2.5 Fault-Tolerance in MCS

Software safety standards such as ISO26262 in addition to specifying criticality requirements also specify reliability requirements that must be satisfied by the system [53]. In MCS, the critical part of the system must be able to function even when faults occur. Faults can be classified [54] according to their duration into: transient, permanent, or intermittent. We focus here on transient and permanent faults affecting the processor. Other types of faults, such as intermittent faults [55], network [56] and memory [57] errors are assumed to be dealt with by other mechanisms.

Transient faults, or soft errors, occur when environmental radiation causes voltage spikes in digital circuits [57]. These faults are typically very short in nature but they have the capacity to cause errors in the system if they propagate to the output. Transient faults are increasing due to the increasing number of transistors and shrinking device sizes which allow smaller charges to cause faults [58]. In order for the system to execute correctly in the presence of transient faults, the system must first detect these faults and then a corrective measure must be taken. Examples of detection mechanisms are lockstep execution [59] and acoustic sensors [60].

In Chapter 5, we propose a mixed-criticality task model and analysis method for fault-tolerant MCS targeting specifically transient faults. The model as-

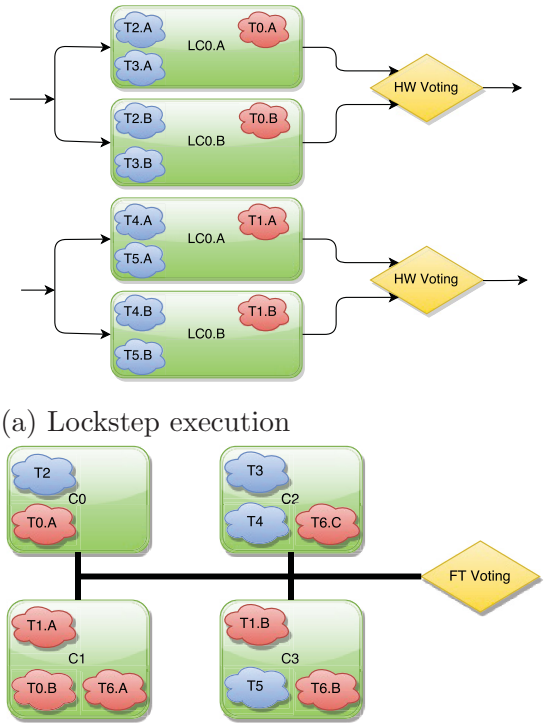
sumes the system's cores are capable of detecting these faults. The model and analysis are independent of the fault detection mechanisms and only require the existence of such mechanisms. Afterwards, we develop a design space exploration approach based on the proposed model which we apply to the common fault-tolerance mechanisms discussed next.

2.5.1 Lockstep execution and on-demand redundancy

Lockstep (LS) execution [59] is a popular method for error detection in automotive Electronic Control Units (ECUs) [61, 62, 63]. LS implements fine-grained redundancy: each store instruction is compared in hardware between the cores before being released to the bus. Cores are paired and each executes the same code in parallel. The results are compared through hardware voting to detect if any errors occurred. Fig. 2.4a shows an example of lockstep execution, where the four physical cores operate essentially as two logical nodes regardless of the workload: both non-critical tasks (blue) and critical tasks (red) must execute on two cores at all times. Correction can be implemented with three processors by majority vote. Lockstep cores are difficult to build and scale due to the precise synchronization required [59, 64].

Alternatively, On-demand Redundancy (ODR) [65, 66] enables more flexibility in system design by allowing cores to be dynamically coupled and decoupled when needed. This allows redundant tasks to be arbitrarily scheduled [65]. Fig. 2.4b shows how ODR allows greater flexibility in scheduling. For example, cores C1, C2, and C3 can be dynamically coupled to execute task T_6 or decoupled to execute other tasks in the system. Non-critical tasks such as T_2

can be executed on a single physical core.



(a) Lockstep execution

(b) On-demand redundancy

Fig. 2.4: Architectures for multicore fault-tolerant systems.

In our design space exploration framework (discussed in Chapter 5), we support four fault-tolerance mechanisms (LS and 3 flavours of ODR), as in Fig. 2.5. In (a), LS execution occurs when a core has internal mechanisms for detecting but not correcting errors. An error simply results in a re-execution on that core. In (b), Dual Modular Redundancy (DMR) replicates a thread on two cores that cannot detect errors by themselves. The task must be re-executed if the executions do not match according to some external comparison or voting mechanism. In (c), Triple Modular Redundancy (TMR) replicates a thread on three cores that cannot detect errors. If an error occurs, the majority

answer is taken from the three replicas and no re-execution is required. Finally, in (d), Passive Replication (PR) is similar to TMR but the final replica does not execute if the first two copies return the same result.

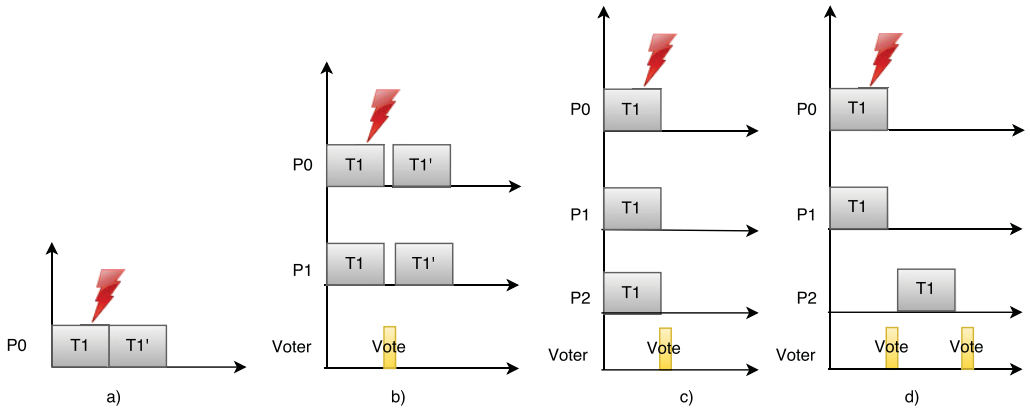


Fig. 2.5: Supported fault-tolerance mechanisms

2.5.2 Transient faults in MCS: related work

A few recent works have begun exploring scheduling MCS considering transient faults. For instance, in [67] re-execution slots are reserved for all tasks (both LO and HI). Since faults are rare events, such over-provisioning wastes CPU time and increases hardware costs. The work we propose in Chapter 5 starts the system in a mode with no over-provisioning, and only considers re-execution for HI-criticality tasks and in response to faults. In [53], the reliability requirements of tasks are accounted for by deriving new values for the HI-criticality tasks' WCETs in the HI mode. However, the work does not model the basic MCS premise that HI tasks can have different WCETs coming from, for example, the designer and the certification authority. The

work in [68] addresses this issue and is similar to our work in considering both fault tolerance and certification requirements for MCS. However, it uses the standard two-mode model of MCS, and LO-criticality tasks are immediately dropped once either a fault or an overrun occurs, compromising the Quality of Service (QoS) provided to LO tasks. We define QoS as the fraction of LO tasks scheduled in a given mode to the total number of LO tasks. In our work, LO task QoS is considered as part of the design problem and optimized. Our work also covers a wider range of fault tolerance mechanisms as opposed to just re-execution on the same core as in [68].

Due to the increasing complexity of designs and richness of architectural options, automated Design Space Exploration (DSE) approaches have flourished in recent years. They have been used in several domains and for various use cases. Examples of such use cases are: [69] which uses a model-driven framework for guided DSE with hints to suppress the search space, applied to a cloud infrastructure configuration problem as a case study, and DESERT tool applied in [70] for component-based model synthesis in vehicle development. [71] provides a taxonomy and review of 188 approaches.

For the specific case of MCS, mapping strategies and automated design space exploration for multicore MCS were first explored in [72]. A two-stage process of nested genetic algorithms is used to solve the resource allocation problem in two distinct phases: a fault tolerance mechanism is selected for each task followed by the allocation of all tasks and necessary replicas to cores. The task model used is simplified, and there is no formal MCS schedulability analysis. [73] considers mapping and scheduling MCS under recommendations

from reliability studies such as zonal and fault hazard analysis. Critical task re-executions take priority over non-critical tasks optimistically scheduled at the same time. [74] performs joint optimization of energy and fault-tolerance when mapping mixed-criticality applications on a Multiprocessor System-on-chip (MPSoC). A genetic algorithm is proposed to explore different hardening techniques and task mappings, and decide the droppable task set. These two works, however, do not model the basic MCS premise of varying WCET estimates as we do in Chapter 5. Architecture-level partitioning techniques such as those described in ARINC 653 standard or in [75] have been used to limit fault propagation for combinations of critical and non-critical functionalities, however, they are outside the scope of this thesis.

2.5.3 Permanent faults in MCS: related work

In [76], task migration for surviving permanent faults is discussed for systems consisting of hard real-time and soft real-time tasks. EDF scheduling is assumed for the hard real-time tasks and a Constant Bandwidth Server (CBS) is used for soft ones. An online greedy algorithm handles task migration and adjusts CBS parameters accordingly. Our work in Chapter 5 uses a more contemporary MCS task model and targets a more general design problem offline. The work in [77] proposes an algorithm to handle task relocation in the case of processor failures. This work is based on a different MCS model (zero slack scheduling [78]) than the one we use. Zero slack scheduling has difficultly incorporating sporadic tasks and even for periodic tasks, ensuring safe execution is not trivial [12].

In contrast, in Chapter 5 we propose an offline design space exploration approach that adds additional degrees of freedom by considering priorities, the initial allocation, and relocations as part of the design problem. The other works mentioned in this section only consider relocation. With our approach, systems that are more optimized in terms of reliability and cost can be designed as reliability is pro-actively taken as an important pillar in the initial design of the system.

2.6 Model-Based Design Using Synchronous Reactive Models

Model-based design [79] is emerging as a solution for handling the increasing complexity of embedded systems. In model-based design, functionality is specified according to a language based on a formal model of computation. After the functional model is validated/verified, a software implementation on the selected platform is automatically generated using methods and tools that guarantee the preservation of semantic properties of interest. This design approach enables advanced verification, reduces design errors, and improves turnaround times for complex designs.

Synchronous Reactive (SR) models are widely used as the formal model of computation in designing control-dominated applications. They have been developed since the early 80s, including Esterel [80], Lustre [81], SIGNAL [82], and the Simulink graphical language [83]. They are popular today in many application domains, such as the avionics and automotive industries. We focus on SR models in this thesis, more specifically on the problem of *design synthesis*

or *implementation synthesis*, i.e., algorithms for turning complex functional specifications into correct and optimal software implementations on embedded platforms.

The traditional approach to implementing synchronous programs such as those written in Esterel or Lustre is by using a single-threaded executable that runs according to an event server model. Reactions to events are decomposed into atomic actions that are partially ordered using a causality analysis of the program. The scheduling is generated at compile time to exploit the partial causality order of functions, and the generated code executes without the need of an operating system. The main concern is to check that the synchronous assumption holds, i.e., ensuring the longest chain of reactions to any event is completed within the task period. In commercial code generators for Simulink, e.g., Simulink Coder and Embedded Coder from MathWorks, or TargetLink from dSPACE, two options are available: a single-task (executing at the base period), or a fixed-priority multi-tasking implementation where one task is generated for each period in the model, with Rate Monotonic (RM) priority assignment. Control systems are *multi-rate* (composed of cooperating functions executing at different periods). In such cases, multi-tasking implementations are preferable, because they allow higher resource utilizations and improve schedulability, compared to their single-task counterparts.

The work in Chapter 6 of this thesis considers the problem of optimizing the multi-task implementation of mixed-criticality SR models. Our target is to synthesize an efficient implementation that preserves the model semantics. We will first discuss the SR model semantics in more depth.

2.6.1 SR semantics preservation

An SR models can be represented by a Directed Acyclic Graph (DAG) $\mathcal{G} = \{\mathcal{N}, \mathcal{E}\}$, where $\mathcal{N}$ is the set of nodes (in the terminology of Lustre, or blocks in Simulink), and $\mathcal{E}$ is the set of edges between the nodes. $\mathcal{N} = \{N_1, \ldots, N_{|\mathcal{N}|}\}$ is the set of *functional nodes*. Each node is triggered by sporadic or periodic events, therefore each node N_i is characterized by a period T_i. Node N_i has one or more *input ports* and *output ports*. $\mathcal{E} = \{E_1, \ldots, E_{|\mathcal{E}|}\}$ is the set of *edges*. An edge (or link) $E_{ij} = (N_i, N_j)$ connects the output port of node N_i (the source node) to an input port of node N_j (the sink). If the edge E_{ij} has no delay on it, called a *feedthrough* edge, then there exists a precedence constraint $N_i \rightarrow N_j$, enforced by assigning higher priority to N_i than N_j. If the edge E_{ij} has a *unit delay* on it, denoted as $N_i \stackrel{-1}{\rightarrow} N_j$, then there exists a *Rate Transition (RT)* block (based on Simulink terminology) inserted between N_i and N_j to break the precedence constraint. A high-rate-to-low-rate dependency edge is an edge E_{ij} connecting a faster writer N_i to a slower reader N_j. A low-rate-to-high-rate dependency edge is an edge E_{ij} connecting a slower writer N_i to a faster reader N_j. The RT block behaves like a *Zero-Order Hold* block for high-rate-to-low-rate dependency edges, or a Unit Delay block plus a Hold block (Sample and Hold) for low-rate-to-high-rate dependency edges [15]. (It is possible to have more than one delay on an edge in general, but we do not consider it here. Note that the term *delay* is used to refer to Rate Transition blocks, based on terminology from signal processing literature, and the term *latency* is used to refer to the real-time latency, measured in milliseconds or seconds). A unit delay on a high-rate-to-low-rate dependency edge does not add additional

dataflow latency, but a unit delay on a low-rate-to-high-rate dependency edge adds latency equal to one period of the low-rate task. *We assume that unit delays are added on all high-rate-to-low-rate dependency edges to maintain the synchronous semantics, but a unit delay may or may not be added on a low-rate-to-high-rate dependency edge, depending on task parameters.* The set of topological dependencies implied by feedthrough edges define a partial order of execution among nodes, which must be respected in both simulation and runtime execution.

Let $N_i(k)$ represent the k-th instance of node N_i, which is periodically activated; and let $r_i(k)$ represent the activation time of $N_i(k)$. Given time $t \geq 0$, we define $n_i(t)$ to be the number of times that N_i has been activated before or at t. For an edge $N_i \rightarrow N_j$, if $i_j(k)$ denotes the input of the k-th occurrence of N_j, then this input must be equal to the output of the last occurrence of N_i, denoted by $o_i(m)$, which is no later than $r_j(k)$:

$$i_j(k) = o_i(m), \text{ where } m = n_i(r_j(k)). \tag{2.19}$$

The timeline on the bottom of Figure 2.6 illustrates the execution of two tasks with SR semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the nodes are activated and compute their outputs from the input. In Figure 2.6, it is $i_j(k) = o_i(m)$ and $i_j(k+1) = o_i(m+2)$.

On the other hand, if $N_i \stackrel{-1}{\rightarrow} N_j$, then the previous output value is read, that is,

$$i_j(k) = o_i(m-1), \text{ where } m = n_i(r_j(k)). \tag{2.20}$$

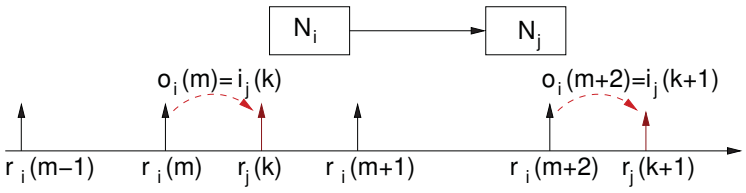


Fig. 2.6: Input/output relation with no edge delay.

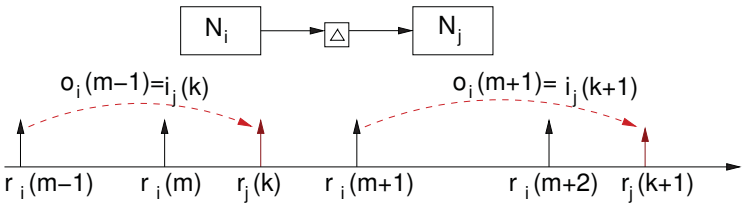


Fig. 2.7: Input/output relation with unit edge delay.

Figure 2.7 shows the effect of the added delay on a high-rate-to-low-rate dependency edge on the input/output relation. Any such added delay helps to relax the precedence constraints imposed on the scheduling and improve schedulability, but causes an increase in end-to-end latency, which in turn causes degraded system control performance; it also increases memory consumption due to duplicated buffers on the edge. Since the edge buffers are typically not large, the effect of increased memory consumption is relatively minor. If we focus on system control performance, it is possible to associate to each delay value a performance cost [84] by using simulation to measure the (application-specific) control error with different delay values on each specific edge. We will illustrate this concept with an example.

Impact of delays on control performance: example

We use a relatively complex example from the Simulink library to illustrate the impact of added delays on control performance. Figure 2.8 shows

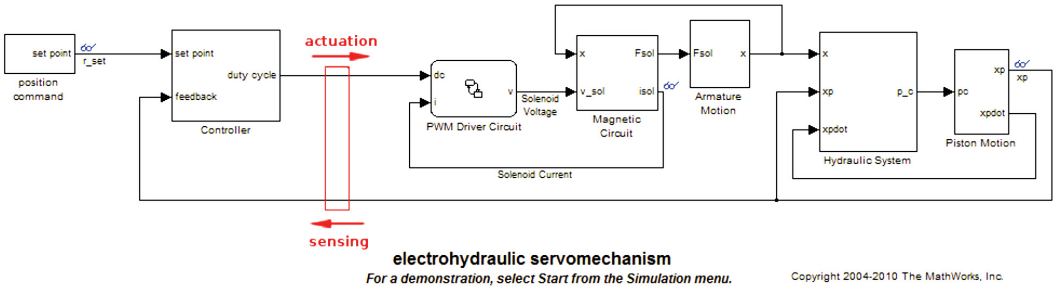


Fig. 2.8: A Simulink example of an hydraulic servomechanism (representative of a suspension control).

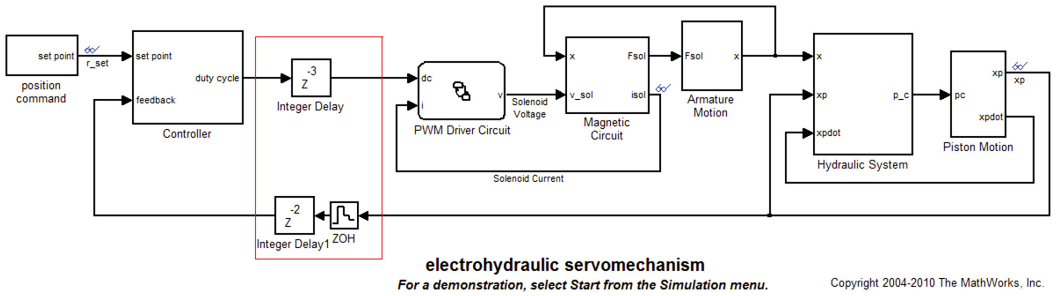


Fig. 2.9: Hydraulic servo with additional communication delays.

a Simulink model of a hydraulic servomechanism controlled by a pulse-width modulated (PWM) solenoid. It is representative of feedback control loops in which there is a flow of data from the sensor to the control and from the control back to the actuator (represented by the red line over the communication links in the figure).

We change the model to add functional delays to the data communications between the sensors, the actuators and the control, to simulate the effect on the quality of control. Figure 2.9 shows the same hydraulic servo model, with additional functional delays on the sensor and actuator paths.

Figure 2.10 shows the results of the simulation of the example model, in

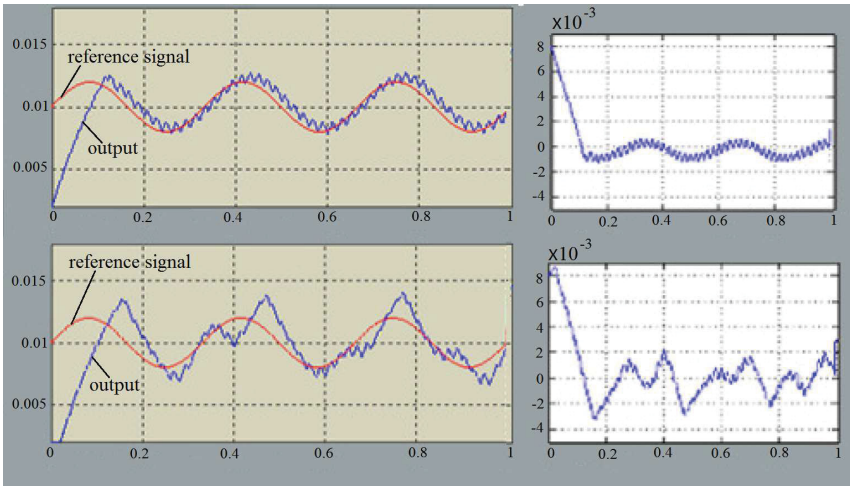


Fig. 2.10: Actuator position and error for the hydraulic servo without (top) and with (bottom) delays.

the top row without added delays, and in the bottom row when a unit delay is added on the sensor path. The results of the simulation show the displacement compared with the reference signal (red line of the left graph), the output (blue line of the left graph), and the error (right graph) in these two cases. The control quality with a unit delay is somewhat degraded, and the simulation results show that the error is now about four times larger. For this control model, it is possible to measure the control error (in millimeter, mm) on the given reference signal for different delay values on the actuator and sensor paths. The results are shown in Table 2.1.

sensor delay	actuator delay	max error (mm)
0	0	0.75
1	0	1.1
0	1	1.6
1	1	1.6

Table 2.1: Max errors for delays on the sensor and actuator paths

In this case, if the control error is the performance parameter of interest,

it is possible to associate to each delay value a performance cost. If convex optimization is used to compute an optimal design configuration, we need to approximate the dependency of the performance from the number of delays with a convex function. For example, we can find a convex hull or linearize the function expressed by the table using a least square approximation, or use continuous piecewise linear function to approximate such curves (approximate formulations for use in mixed integer linear programming (MILP) can be found in [85]).

In this thesis, we assume that *a linear function is given which approximates the relationship between the delays on the communication links and the control error*. We adopt the optimization objective of minimizing the weighted sum of edge delays, where the weight for each edge is determined by the designer, based on its effect on the control performance and/or buffer size on each edge.

2.6.2 Model-based design: related work

[86] discussed general conditions for a semantics-preserving implementation of communications among synchronous nodes on a uniprocessor platform, and presented a wait-free communication mechanism. [87] addressed the problem of finding an optimal multitask implementation for multirate Simulink models with fixed-priority scheduling, and presented a branch-and-bound algorithm to minimize the memory usage from communication buffers. [15] presented a more general formulation of the problem in [87], and an MILP formulation. [84] addressed the problem of minimizing the use of functional delays in implementation of SR models based on EDF scheduling.

For MCS, [88] presented algorithms for static time-triggered scheduling of a mixed criticality taskset, equivalent to a *single-rate* SR model, on a uniprocessor. First they assign priorities to jobs with Own-Criticality-Based Priority (OCBP) [43], then they construct two separate schedule tables, S_{LO} for LO-criticality mode and S_{HI} for HI-criticality mode. At runtime, they switch from S_{LO} to S_{HI} upon the mode switch event from LO to HI criticality mode. [89] presented algorithms for static time-triggered scheduling of a *multi-rate* mixed-criticality synchronous program on a uniprocessor. First, the multi-rate block diagram is unrolled into a single-rate block diagram within its hyper-period. Then, OCBP is applied to assign priorities to nodes in the unrolled block diagram. Finally, the techniques in [88] are applied to construct schedule tables. Time-triggered scheduling may be suitable for certain application domains such as avionics, but is not a common practice in cost-sensitive application domains like automotive systems. In contrast, Chapter 6 presents an efficient method for implementing SR models for MCS using dynamic scheduling.

Chapter 3

Task Allocation in Multicore Real-Time Systems

Multicore architectures are becoming increasingly common in high performance real-time applications. One of the main challenges introduced by multicore architectures compared to their single core counterparts is finding efficient solutions to the NP-hard task allocation problem. Many of the previous works on task allocation in real-time systems ignored the effect of shared resources. Shared resources can introduce significant blocking, making systems where task allocation ignored their effect unschedulable or inefficient [90].

In this chapter, we propose novel approaches to solve the task allocation problem in multicore real time systems more efficiently. The selection of shared resources protection mechanisms is considered as an additional design variable. We exploit the tradeoff between lock-based protection mechanisms (which introduce blocking and require no additional memory), and waitfree methods

(which introduce negligible blocking but require additional memory cost). We target both suspension-based locking protocols (Multiprocessor Priority Ceiling Protocol (MPCP)) and spin-based locking protocols (Multiprocessor Stack Resource Policy (MSRP)). Our objective is to find a system configuration (task allocation, selection of resource mechanisms) that is (1) schedulable, and (2) requires minimal memory for wait-free buffers. We solve this problem using the following approaches:

- We develop a Mixed Integer Linear Programming (MILP) formulation for task allocation, priority assignment, and selection of resource protection mechanisms (between MPCP and wait-free or between MSRP and wait-free), while finding schedulable systems with optimal cost (Section 3.3).

- To address the limitations of ILP when scaling to large designs, we propose two heuristic algorithms (Section 3.4): (1) GS-WF (greedy slacker with wait-free), a baseline algorithm extending the lock-based greedy slacker (GS) [90] partitioning algorithm with wait-free support, and (2) MPA (memory-aware partitioning algorithm). Experiments show that both algorithms increase schedulability, with MPA generally outperforming GS-WF.

Experimental results indicate the effectiveness of the proposed work in significantly extending the range of systems that can be scheduled on a particular hardware platform, enabling systems in the 76%-88% utilization range to be scheduled despite not being schedulable with previous resource-aware task allocation algorithms.

The rest of this chapter is organized as follows. In Section 3.1, we review our system model. In Section 3.2, we present an overview of existing resource-aware task allocation algorithms. In Sections 3.3, and 3.4, we present the two solution approaches (MILP formulation and heuristic algorithms). In Section 3.5, we present our experimental results. Finally, Section 3.6 concludes the chapter.

3.1 System Model

The system model is based on the basic model presented in Section 2.1. We will briefly review it in this section. The system under consideration consists of m cores $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$. n tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots \tau_n\}$ are statically allocated to them and scheduled by static priorities. Tasks share a set of q resources $\mathcal{R} = \{r_1, r_2, \ldots r_q\}$. Each task τ_i is activated by a periodic or sporadic event stream with T_i as the period or minimum interarrival time. The execution of τ_i is composed of a set of alternating *critical sections* and *normal sections*. In the latter, τ_i executes without using a (global or local) shared resource. The worst case execution time (WCET) is a tuple $\{C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,s(i)-1}, C_{i,s(i)}\}$, where $s(i)$ is the number of normal sections, and $s(i) - 1$ is the number of critical sections. $C_{i,j}$ ($C'_{i,j}$) is the WCET of the j-th normal section (critical section) of τ_i. We use parentheses around the resource to refer to the critical section accessing it. For example, $C'_{i,(r1)}$ denotes the WCET of τ_i's critical section accessing resource r_1. The set of resources accessed by τ_i is denoted as $\mathcal{R}(i)$. π_i is the nominal priority of τ_i (*the higher the number, the lower the priority*), and P_i is the core where τ_i executes. The shared resource accessed by the j-th critical section of τ_i is $\mathcal{S}_{i,j}$.

τ_i must finish its execution within its relative deadline D_i ($\leq T_i$). That is, a task is said to be schedulable if the response time $R_i \leq D_i$. Task utilization is defined as $u_i = \frac{C_i}{T_i}$.

3.2 Resource-Aware Task Allocation

In [14], Lakshmanan et al. made the observation that resource-agnostic task allocation algorithms can introduce bottlenecks in system design by placing tasks that share resources on different cores. This can lead to an increase in blocking time. In [14], a task allocation heuristic called SPA tailored to the MPCP protocol is also presented. It organizes tasks that share resources into groups in order to assign them to the same processor. Nemati et al. [91] present another partitioning heuristic for MPCP named BPA, which tries to identify communication clusters such that globally shared resources are minimized and the blocking time is reduced.

The first work on the partitioning problem with spin locks (MSRP) is [90]. Two solutions are proposed, one is an ILP formulation that can provide the optimal solution, but with exponential complexity. The other is the GS [90] heuristic, which obtains good quality solutions in a much shorter time. GS assigns tasks to cores sequentially according to a simple greedy policy that tries to maximize the least slack (i.e., the difference between the deadline and the worst case response time). Our work targets a similar problem, with two notable differences: one is considering suspension locks (MPCP) in addition to spin locks (MSRP), and the other is the selective use of wait-free methods. In [19], the CASR (Communication Affinity and Slack with Re-

tries) algorithm is proposed which also performs task allocation for systems scheduled by MSRP. In this chapter, we experimentally compare our proposed algorithms with SPA, BPA when applied to MPCP. We also apply GS and CASR to MPCP and compare our algorithms to it. For MSRP, we compare with GS and CASR.

3.3 Problem Formulation with MILP

Lock-based mechanisms provide data consistency for shared resources at the price of blocking time, which negatively impacts schedulability. Wait-free methods effectively experience no blocking which makes them an attractive option for improving schedulablity, but require additional memory for buffer replicas. We propose to use a combination of lock-based and wait-free mechanisms to leverage their complementary characteristics. In this section, we present a MILP formulation that finds a schedulable system configuration (if one exists) with the minimum memory cost. The design variables are (a) the assignment of tasks to processes; (b) the assignment of task priorities; and (c) the selection of protection mechanisms (between MSRP and wait-free or between MPCP and wait-free) for resources.

3.3.1 General system variables and constraints

We first discuss the set of variables in the formulation. The task allocation is denoted by a set of binary variables $A_{i,p}$, defined as 1 if task τ_i is allocated

to core p and 0 otherwise. A task shall be allocated to one and only one core:

$$\forall \tau_i : \sum_{p=1}^{m} A_{i,p} = 1. \tag{3.1}$$

A binary variable $G_{i,j}$ is defined as 1 if tasks τ_i and τ_j are assigned to the same core and 0 otherwise. The constraints in Eqs. (3.2) and (3.3) guarantee the consistency of $G_{i,j}$, $A_{i,p}$, and $A_{j,p}$ for all cores p.

$$\forall \tau_i \neq \tau_j, \forall p : G_{i,j} \geq A_{i,p} + A_{j,p} - 1, \tag{3.2}$$

$$\forall \tau_i \neq \tau_j, \forall p_1 \neq p_2 : G_{i,j} \leq 2 - A_{i,p_1} - A_{j,p_2}. \tag{3.3}$$

For any two tasks τ_i and τ_j, the binary variable $\pi_{i,j}$ is set to 1 if τ_i has a higher priority than τ_j and 0 otherwise. The antisymmetry and transitivity of priority orders shall be enforced through the following constraints:

$$\forall \tau_i \neq \tau_j : \pi_{i,j} + \pi_{j,i} = 1 \tag{3.4}$$

$$\forall \tau_i \neq \tau_j \neq \tau_k : \pi_{i,k} \geq \pi_{i,j} + \pi_{j,k} - 1 \tag{3.5}$$

The binary variable W_r is defined as 1 if the resource r is protected by a wait-free mechanism and 0 if it is protected by a lock-based mechanism. Local resources usually do not cause much blocking (a maximum of one critical section). Hence, to simplify the problem, we use SRP for all local resources. Let $rd(r)$ be the set of tasks reading from a shared resource r, $wt(r)$ be its writer, and $rw(r)$=$rd(r) \cup wt(r)$. The following constraint forces W_r to be 0 if the all readers of resource r are assigned to the same core as the writer ($G_{i,wt(r)} = 1$, $\forall \tau_i \in rd(r)$):

$$\forall r : W_r \leq |rd(r)| - \sum_{\tau_i \in rd(r)} G_{i,wt(r)}. \tag{3.6}$$

The schedulability of the system requires that each task response time is no longer than its deadline:

$$\forall \tau_i : R_i \leq D_i. \tag{3.7}$$

A sufficient schedulability condition is that the total utilization on each core cannot exceed 1:

$$\forall p : \sum_{i=1}^{n} (A_{i,p} \cdot \frac{C_i}{T_i}) \leq 1. \tag{3.8}$$

This constraint is effective in quickly ruling out trivially infeasible task allocations.

3.3.2 MSRP response time formulation

To calculate the MSRP response time, we first rewrite Eq. (2.15) by substituting (2.13) for C^*:

$$\begin{aligned} R_i = & C_i + \underbrace{\max\{B_i^l, B_i^r\}}_{BL_i} + \sum_{\pi_h < \pi_i \wedge P_h = P_i} \left\lceil \frac{R_i}{T_h} \right\rceil C_h \\ & + \underbrace{\sum_{1 \leq j < s(i)} L_{i,j} + \sum_{\pi_h < \pi_i \wedge P_h = P_i} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot \sum_{1 \leq k < s(h)} L_{h,k}}_{BH_i}. \end{aligned} \tag{3.9}$$

The response time R_i is now the sum of four terms. The *first* term is the task WCET (a constant). The *second* term, BL_i, represents the blocking time that may be experienced by the task before it can start executing, caused by lower priority tasks on the same core accessing a local or a remote resource. Its formulation is detailed next. The *fourth* term, BH_i, is the spin time on global resources to which either the task τ_i itself or local high priority tasks try to access. Its formulation is discussed afterwards.

The *third* term is the interference from local higher priority tasks. An integer variable $OL_{i,h}$ is introduced to represent the number of times task τ_h may interfere with τ_i.

$$\forall \tau_h \neq \tau_i : OL_{i,h} = \left\lceil \frac{R_i}{T_h} \right\rceil \cdot G_{i,h} \cdot \pi_{h,i} \tag{3.10}$$

This can be linearized as:

$$\begin{cases} OL_{i,h} \geq \dfrac{R_i}{T_h} - M(1 - G_{i,h}) - M(1 - \pi_{h,i}) \\ OL_{i,h} < 1 + \dfrac{R_i}{T_h}, \quad OL_{i,h} \leq M \cdot G_{i,h}, \quad OL_{i,h} \leq M \cdot \pi_{h,i} \end{cases}$$

where M is a large constant, such as D_i.

Formulation of BL_i

From Eqs. (2.11) and (2.14), BL_i is the maximum over $BL_{i,r}$ for all resources, where $BL_{i,r}$ is the maximum blocking time encountered by task τ_i due to an access to resource r by a local lower priority task:

$$\forall \tau_i, \forall r \in \mathcal{R} : BL_i \geq BL_{i,r}. \tag{3.11}$$

$BL_{i,r}$ depends on the type of resource. There are two cases.

Case 1: r is a local resource. As in Eq. (2.11), the following conditions must be satisfied to include the access to r in τ_i's blocking calculation: r is local; r is accessed by a lower priority task τ_j allocated to the same core as τ_i; and, r has a ceiling greater than or equal to the priority of τ_i. These conditions are formulated by the following constraints:

$$\forall r, \quad \forall \tau_i \neq \tau_j \wedge \tau_j \in rw(r), \quad \forall \tau_h \neq \tau_j \wedge \tau_h \in rw(r) :$$

$$BL_{i,r} \geq C'_{j,(r)} - M|i - wt(r)|(1 - G_{i,wt(r)}) - M(1 - G_{i,j})$$

$$-M(1 - \pi_{i,j}) - M(1 - G_{j,h}) - M|i - h|(1 - \pi_{h,i}).$$

The constraint is effective (i.e., $C'_{j,(r)}$ is accounted for in $BL_{i,r}$'s calculation) only if the writer of r is on the same core ($G_{i,wt(r)} = 1$), τ_j is on the same core ($G_{i,j} = 1$) and has lower priority ($\pi_{i,j} = 1$), τ_h is on the same core, and finally r has a ceiling that is either higher than τ_i's priority ($\pi_{h,i} = 1$) or equal to it (if $h = i$ and hence τ_i accesses r).

Case 2: r is a global resource. If r is protected with a lock-based mechanism ($W_r = 0$), as in Eq. (2.14), τ_i can be blocked by (**2a**) local lower priority tasks accessing r, or (**2b**) remote tasks accessing r provided that there is at least one local low priority task accessing it. Since accesses to global resources are completed in a FIFO manner with at most one concurrent request from each core, we split $BL_{i,r}$ on a per-core basis and then sum over all cores:

$$\forall \tau_i, r, p : BL_{i,r} \geq \sum_{p=1}^{m} BL_{i,r,p} - M \cdot W_r. \tag{3.12}$$

In case **2a**, three conditions must be satisfied: r is global; r is accessed by a task τ_j allocated on processor p; and τ_j is on the same core p as τ_i with a lower priority than τ_i. Eq. (3.13) formulates the conditions: it is effective only if both τ_i and τ_j are allocated on p ($A_{i,p} = 1$ and $A_{j,p} = 1$), τ_j has lower priority than τ_i ($\pi_{i,j} = 1$), and there exists a task τ_k on a different core that accesses the resource ($G_{j,k} = 0$; resource r is global). In addition, if τ_j is not the writer, then the writer and τ_j are allocated to different cores.

$$\begin{aligned} &\forall r, \quad \forall p, \quad \forall \tau_i \neq \tau_j \neq \tau_k \wedge \tau_j \in rw(r) \wedge \tau_k \in rw(r) : \\ &BL_{i,r,p} \geq C'_{j,(r)} - M(1 - A_{i,p}) - M(1 - A_{j,p}) \\ &-M(1 - \pi_{i,j}) - M \cdot G_{j,k} - M|j - wt(r)| \cdot G_{j,wt(r)} \end{aligned} \tag{3.13}$$

For case **2b**, it shall be that r is global; r is accessed by a remote task τ_j allocated on p; and there exists a lower priority task τ_k on the same core as

Table 3.1: Task parameters for the ILP example system

Task	Period (ms)	WCET (ms)	Core	Priority	r_1 access time
τ_1	500	100.0	p_1	4	-
τ_2	15	0.2	p_1	3	0.10
τ_3	200	5.0	p_2	2	0.50
τ_4	60	5.0	p_2	1	0.15

τ_i that accesses r. This can be achieved with Eq. (3.14), where the last two terms ensure that r is global, similar to Eq. (3.13).

$$\begin{aligned} \forall r, \quad \forall p, \quad \forall \tau_i \neq \tau_j \neq \tau_k \wedge \tau_j \in rw(r) \wedge \tau_k \in rw(r) : \\ BL_{i,r,p} \geq C'_{j,(r)} - M(1 - A_{j,p}) - M \cdot G_{i,j} \\ -M(1 - G_{i,k}) - M(1 - \pi_{i,k}) \\ -M|j - wt(r)| \cdot G_{j,wt(r)} - M|k - wt(r)| \cdot G_{k,wt(r)} \end{aligned} \tag{3.14}$$

Formulation of BH_i

The analysis in [17] over-estimates BH_i, as shown in [38]. Consider the task set in Table 3.1, where tasks τ_2, τ_3, and τ_4 share a global resource r_1. τ_1 is allocated to core p_1 along with one high priority task (τ_2). BH_1, as in Eq. (3.9), consists of τ_1's remote accesses and remote accesses by higher priority tasks. Since τ_1 does not access any remote resources, we only need to account for τ_2's remote blocking. $L_{2,1}$ is the spin time encountered by task τ_2 to access resource r_1. Using Eq. (2.12), $L_{2,1}$ is the sum of the longest accesses to r_1 from each other core. In the example, we have only one other core (p_2) and the longest critical section has a length of 0.5, hence $L_{2,1} = 0.5$, and

$$BH_1 = \sum_{1 \leq j < s(1)} L_{1,j} + \left\lceil \frac{R_1}{T_2} \right\rceil \sum_{1 \leq k < s(2)} L_{2,k} = 0 + \left\lceil \frac{R_1}{15} \right\rceil \cdot 0.5.$$

The iterative procedure to compute R_1 can start from $C_1 = 100$. $BH_1 = \lceil \frac{100}{15} \rceil \cdot 0.5 = 3.5$, and R_1 converges at 103.5.

In this example, the analysis in [17] essentially estimates that τ_2 would try to access r_1 seven times during a period of 100 time units and each time, it finds that a task from p_2 is accessing it. However, this is pessimistic as a maximum of five instances of tasks on p_2 can access r_1 (two instances of τ_3 and three instances of τ_4). Therefore, a tighter bound for BH_1 can be found by taking the minimum of the number of local accesses and the number of remote accesses to the global resource, in this case $BH_1 = 0.5 \cdot \min(7, 5) = 2.5$.

The analysis can be further tightened: we may identify the accessing task and include its access time in the calculation instead of pessimistically taking the longest critical section on each processor. For the example, we would have a maximum of two accesses from τ_3 and three from τ_4, hence $BH_1 = 0.5 \cdot 2 + 0.15 \cdot 3 = 1.45$.

Taking advantage of these optimizations, we formulate BH. The is done by breaking BH down by resource and then by task. $BH_{i,r,j}$ is the blocking time encountered by τ_i when it tries to access global resource r and is forced to wait because remote task τ_j is accessing r. For example, for remote task τ_4, $BH_{1,1,4} = 0.15 \cdot \min(3, 7 - 2)$. The actual number of accesses (**three** in this case) is thus affected by:

1. Remote overlap: the maximum number of possible overlaps between τ_j and τ_i (three in the example), denoted by the variable $OR_{i,j}$;

2. Local accesses: the number of times τ_i or local higher priority tasks attempt to access the resource (seven in the example), denoted by the variable $NH_{i,r}$;

and,

3. Local accesses already accounted for. This term comes from the longer critical sections of tasks that are allocated on τ_j's processor (in the example, two from τ_3 since its critical section is longer $0.5 > 0.15$), denoted by the variable $NC_{i,r,j}$.

Eq. (3.15) formulates $BH_{i,r,j}$, where the last term ensures that only global resources are considered. The min operator is commonly implemented in ILP solvers to simplify formulations. In general, the operation $z = \min(x, y)$ can be easily translated to ILP by defining an auxiliary binary variable b that is set to 0 if $x \leq y$ and 1 otherwise.

$$\begin{aligned} &\forall \tau_i \neq \tau_j \wedge \tau_j \in rw(r): \\ BH_{i,r,j} \geq\ & C'_{j,(r)} \cdot \min(OR_{i,j}, NH_{i,r} - NC_{i,r,j}) \\ & -M|wt(r) - j| \cdot G_{wt(r),j} \end{aligned} \tag{3.15}$$

BH_i is then calculated by summing $BH_{i,r,j}$ over r and j:

$$\forall \tau_i, \forall r \in \mathcal{R} : BH_{i,r} \geq \sum_j BH_{i,r,j} - M \cdot W_r \tag{3.16}$$

$$\forall \tau_i : BH_i = \sum_{r \in \mathcal{R}} BH_{i,r} \tag{3.17}$$

Finally, we need to formulate constraints to bound the three variables used in $BH_{i,r,j}$'s calculation. The number of overlaps $OR_{i,j}$ between two remotely allocated tasks (hence $G_{i,j} = 0$) is equal to $1 + \left\lceil \frac{R_i}{T_j} \right\rceil$, formulated as:

$$\begin{aligned} &\forall \tau_i \neq \tau_j : \quad 1 + \tfrac{R_i}{T_j} - M \cdot G_{i,j} \leq OR_{i,j} < 2 + \tfrac{R_i}{T_j} \\ &\forall \tau_i \neq \tau_j : \quad OR_{i,j} \leq M(1 - G_{i,j}) \end{aligned} \tag{3.18}$$

$NH_{i,r}$ can be calculated by considering the number of accesses from each task separately:

$$\forall \tau_i, r : NH_{i,r} = \sum_{\tau_j \in rw(r)} NH_{i,r,j} \tag{3.19}$$

$NH_{i,r,j}$ is given by the following constraint (where the last term ensures only global resource accesses are considered):

$$NH_{i,r,j} \geq OL_{i,j} - M|wt(r) - j| \cdot G_{wt(r),j}.$$

Finally, $NC_{i,r,j}$ represents accesses to r by tasks that are allocated on τ_j's core and have longer critical sections. We split $NC_{i,r,j}$ on a per task basis:

$$\forall r, \forall \tau_i \neq \tau_j \wedge \tau_j \in rw(r) : NC_{i,r,j} = \sum_{\tau_k \in rw(r), \tau_k \neq \tau_j \neq \tau_i} NC_{i,r,j,k},$$

where $NC_{i,r,j,k}$ is the number of accesses to r during τ_i's execution by tasks τ_k that are (a) allocated to τ_j's core and (b) have longer critical sections for accessing r than τ_j. $NC_{i,r,j,k}$ can be tightly bounded by:

$$\forall r, \forall \tau_i \neq \tau_j \neq \tau_k \wedge \tau_j \in rw(r) \wedge \tau_k \in rw(r) \wedge C'_{k,(r)} \geq C'_{j,(r)} : \begin{cases} NC_{i,r,j,k} \geq OR_{i,k} - M(1 - G_{j,k}) - M \cdot G_{k,wt(r)}, \\ NC_{i,r,j,k} \leq M \cdot G_{j,k}, \quad NC_{i,r,j,k} \leq M(1 - G_{k,wt(r)}), \\ NC_{i,r,j,k} \leq OR_{i,k}; \end{cases} \tag{3.20}$$

$$\forall r, \forall \tau_i \neq \tau_j \neq \tau_k \wedge \tau_j \in rw(r) \wedge \tau_k \in rw(r) \wedge C'_{k,(r)} < C'_{j,(r)} : \quad NC_{i,r,j,k} = 0. \tag{3.21}$$

The constraints in Eq. (3.20) are for the case that τ_k accesses the resource r with a longer critical section than τ_j. The first constraint is that $NC_{i,r,j,k}$ is lower bounded by the number of possible accesses of τ_k to r during τ_i's lifetime ($OR_{i,k}$). This condition is enforced only when τ_j and τ_k are on the

same core ($G_{j,k} = 1$) and r is global ($G_{k,wt(r)} = 0$). The rest of (3.20), together with (3.21), ensures that $NC_{i,r,j,k}$ is set to 0 if any of the previous conditions is not satisfied.

3.3.3 MPCP response time formulation

For MPCP, the response time is calculated according to Eq. (2.10). We define a new variable $OL2_{i,h}$ to represent the term $\left\lceil \frac{R_i+R_h-C_h}{T_h} \right\rceil$, a safe upper bound on the number of preemptions by the local higher priority task τ_h. $OL2_{i,h}$ is bounded by the following constraints: $\forall \tau_i \neq \tau_h$,

$$\begin{cases} OL2_{i,h} \geq \dfrac{R_i + R_h - C_h}{T_h} - M(1 - G_{i,h}) - M \cdot \pi_{i,h}, \\ OL2_{i,h} < 1 + \dfrac{R_i + R_h - C_h}{T_h}, \\ OL2_{i,h} \leq M \cdot G_{i,h}, \\ OL2_{i,h} \leq M(1 - \pi_{i,h}). \end{cases} \tag{3.22}$$

We also define two new variables BC_i and BR_i to represent B_i^l and B_i^r respectively. Eq. (2.10) becomes:

$$\forall \tau_i : R_i = C_i + BC_i + BR_i + \sum_{h \neq i} C_h \cdot OL2_{i,h}. \tag{3.23}$$

Local blocking time BC_i

BC_i in MPCP is calculated by Eq. (2.5). In the worst case, each local lower priority task can block τ_i. Hence,

$$\forall \tau_i : BC_i = \sum_{j \neq i} BC_{i,j}, \tag{3.24}$$

where $BC_{i,j}$ is the local blocking suffered by task τ_i due to a resource access by the local lower priority task τ_j. $BC_{i,j}$ must be set to 0 for remote or higher

priority tasks. Also, the formulation of $BC_{i,j}$ differs depending on whether τ_j is a writer for the resource. As in Eq. (2.5), only the longest resource access for τ_j is considered in the calculation.

When τ_j is not the writer of r, we add two constraints to calculate $BC_{i,j}$ (Eq.(3.25)). The first constraint is for the case where r is a local resource ($G_{j,wt(r)} = 1$), and the second when r is a global resource protected by MPCP ($G_{j,wt(r)} = 0$, $W_r = 0$). Both constraints ensure that τ_j is only accounted for if it is a local lower priority task ($G_{i,j} = 1$, $\pi_{i,j} = 1$).

$$\forall r,\ \forall \tau_i \neq \tau_j \wedge \tau_j \in rw(r) \wedge \tau_j \neq wt(r):$$
$$\begin{cases} BC_{i,j} \geq s(i) \cdot C'_{j,(r)} - M(1 - G_{i,j}) - M(1 - \pi_{i,j}) \\ \qquad\qquad -M(1 - G_{j,wt(r)}), \\ BC_{i,j} \geq s(i) \cdot C'_{j,(r)} - M(1 - G_{i,j}) - M(1 - \pi_{i,j}) \\ \qquad\qquad -M \cdot G_{j,wt(r)} - M \cdot W_r. \end{cases} \tag{3.25}$$

When τ_j is the writer, similar constraints are defined for $BC_{i,j}$ (Eq.(3.26)). The difference is that there must exist another task τ_k accessing r. The first constraint is for the case in which r is a local resource ($G_{j,k} = 1$), and the second for the case in which r is global and protected by MPCP ($G_{j,k} = 0$, $W_r = 0$).

$$\forall r,\ \forall \tau_i \neq \tau_j \neq \tau_k \wedge \tau_j = wt(r) \wedge \tau_k \in rw(r):$$
$$\begin{cases} BC_{i,j} \geq s(i) \cdot C'_{j,(r)} - M(1 - G_{i,j}) - M(1 - \pi_{i,j}) \\ \qquad\qquad -M(1 - G_{j,k}), \\ BC_{i,j} \geq s(i) \cdot C'_{j,(r)} - M(1 - G_{i,j}) - M(1 - \pi_{i,j}) \\ \qquad\qquad -M \cdot G_{j,k} - M \cdot W_r, \end{cases} \tag{3.26}$$

Remote priority ceiling

Before formulating remote blocking, we incorporate the concept of remote priority ceiling, used in Eq. (2.6), into the ILP formulation. The remote priority ceiling $\Pi_{i,r1}$ of a global critical section $C'_{i,(r1)}$ under MPCP is given by $\Pi_{base} + \Pi_x$ where Π_{base} is a priority level higher than that of any task in the system, and τ_x is the highest priority task accessing r_1 allocated to any core other than that of τ_i.

To define the relative ordering between the priority ceilings of two global critical sections $C'_{i,(r1)}$ and $C'_{j,(r2)}$, we define a new variable $\Pi_{i,r1,j,r2}$, which is set to 1 if an access to r_2 has no impact on the response time of $C'_{i,(r1)}$. This happens if the global critical section of τ_i accessing r_1 has an equal or higher remote ceiling than the critical section of τ_j accessing r_2. $\Pi_{i,r1,j,r2}$ is 1 when the highest priority task accessing r_1 and allocated to any other core than τ_i's has a higher (or equal) priority than the highest priority task accessing r_2 and allocated to any other core than τ_j's.

To derive $\Pi_{i,r1,j,r2}$, we split the calculation for the remote tasks. Let $\tau_x \neq \tau_i$ be any task (remote to τ_i) that accesses the global resource r_1, and let $\tau_y \neq \tau_j$ be any task (remote to τ_j) that accesses the global resource r_2.

1. $\Pi_{i,r1,j,r2,x,y}$ is 0 if task τ_x has a lower priority than the remote task τ_y and 1 otherwise.

2. $\Pi_{i,r1,j,r2,x}$ is 1 if τ_x has a higher priority than **all** such τ_y (i.e., if all $\Pi_{i,r1,j,r2,x,y} = 1$) and 0 otherwise.

3. $\Pi_{i,r1,j,r2}$ is 1 when $C'_{i,(r1)}$ has a higher (or equal) priority ceiling than $C'_{j,(r2)}$

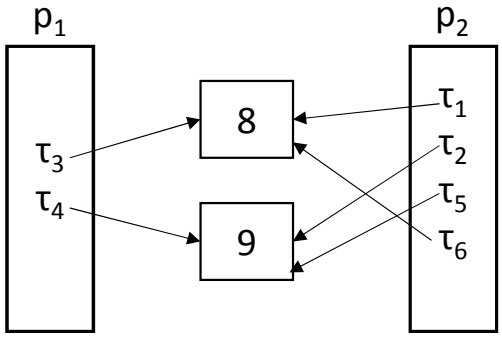


Fig. 3.1: MPCP remote ceiling example

Table 3.2: Remote priority ceiling calculation

Parameter	Value	Parameter	Value	Parameter	Value
$\Pi_{3,8,4,9,1,2}$	1	$\Pi_{3,8,4,9,1}$	1	$\Pi_{3,8,4,9}$	1
$\Pi_{3,8,4,9,1,5}$	1				
$\Pi_{3,8,4,9,6,2}$	0	$\Pi_{3,8,4,9,6}$	0		
$\Pi_{3,8,4,9,6,5}$	0				
$\Pi_{4,9,3,8,2,1}$	0	$\Pi_{4,9,3,8,2}$	0	$\Pi_{4,9,3,8}$	0
$\Pi_{4,9,3,8,2,6}$	1				
$\Pi_{4,9,3,8,5,1}$	0	$\Pi_{4,9,3,8,5}$	0		
$\Pi_{4,9,3,8,5,6}$	1				

if there is **any** τ_x that has a higher priority than all such τ_y (i.e., if any τ_x has $\Pi_{i,r1,j,r2,x} = 1$).

We illustrate these steps with the example in Fig. 3.1. Assume the tasks are indexed by decreasing priority (τ_1 has the highest priority and τ_6 the lowest). We compute the ceiling of resource accesses from core p_1 to resources r_8 and r_9. For τ_3, $\Pi_{3,8,4,9} = 1$ since the remote task τ_1 has a higher priority than any task accessing r_9. Also, $\Pi_{4,9,3,8} = 0$ for the same reason. Table 3.2 shows all the values.

$\Pi_{i,r1,j,r2,x,y}$ is set to 0 if τ_x has lower priority than τ_y. $\Pi_{i,r1,j,r2,x}$ can be viewed as the result of an AND operation on all *valid* $\Pi_{i,r1,j,r2,x,y}$ values. Since the allocation of τ_y is only known at runtime, $\Pi_{i,r1,j,r2,x,y}$ must be invalidated

(set to 1 so that it does not affect the AND operation) if τ_y must not affect the result of the operation (for example, if it is local to τ_j). Similarly $\Pi_{i,r1,j,r2}$ can be viewed as the result of an OR operation on all the valid $\Pi_{i,r1,j,r2,x}$ values. $\Pi_{i,r1,j,r2,x}$ must be set to 0 if it must not affect the result of the OR operation (e.g., if τ_x is local to τ_i).

We enforce the following constraints:

$$\begin{aligned}
&\forall r_1 \neq r_2, \forall \tau_i \wedge \tau_i \in rw(r_1), \forall \tau_j \neq \tau_i \wedge \tau_j \in rw(r_2), \\
&\forall \tau_x \neq \tau_i \wedge \tau_x \in rw(r_1), \forall \tau_y \neq \tau_j \wedge \tau_y \in rw(r_2) : \\
&\begin{cases}
\text{if } \tau_y = wt(r_2) & \Pi_{i,r1,j,r2,x,y} \leq G_{j,y} + \pi_{x,y}, \\
\text{if } \tau_y \neq wt(r_2) & \Pi_{i,r1,j,r2,x,y} \leq G_{j,y} + \pi_{x,y} + G_{wt(r_2),y}, \\
& \Pi_{i,r1,j,r2,x,y} \geq \pi_{x,y}, \\
& \Pi_{i,r1,j,r2,x,y} \geq G_{j,y}, \\
\text{if } \tau_y \neq wt(r_2) & \Pi_{i,r1,j,r2,x,y} \geq G_{y,wt(r_2)}.
\end{cases}
\end{aligned} \tag{3.27}$$

The first two constraints in Eq. (3.27) set $\Pi_{i,r1,j,r2,x,y}$ to 0 only if (a) τ_x has a lower priority than τ_y ($\pi_{x,y} = 0$); (b) τ_y is remote to τ_j ($G_{j,y} = 0$); (c) τ_y writes to global resource r_2 or τ_y remotely reads from r_2 ($G_{wt(r2),y} = 0$). The other three constraints set $\Pi_{i,r1,j,r2,x,y}$ to 1 in the opposite scenarios (if τ_x has a higher priority than τ_y or if τ_y is invalid for the calculation of $\Pi_{i,r1,j,r2}$).

Next, we enforce the following constraints on $\Pi_{i,r1,j,r2,x}$:

$$\begin{aligned}
&\forall r_1 \neq r_2, \forall \tau_i \wedge \tau_i \in rw(r_1), \forall \tau_j \neq \tau_i \wedge \tau_j \in rw(r_2), \\
&\forall \tau_x \neq \tau_i \wedge \tau_x \in rw(r_1), \forall \tau_y \neq \tau_j \wedge \tau_y \in rw(r_2) : \\
&\begin{cases}
& \Pi_{i,r1,j,r2,x} \leq \Pi_{i,r1,j,r2,x,y}, \\
& \Pi_{i,r1,j,r2,x} \leq 1 - G_{i,x}, \\
\text{if } \tau_x \neq wt(r_1) & \Pi_{i,r1,j,r2,x} \leq 1 - G_{x,wt(r_1)}.
\end{cases}
\end{aligned} \tag{3.28}$$

The first and second constraints in Eq. (3.28) ensure respectively that $\Pi_{i,r1,j,r2,x}$ is 0 if any $\Pi_{i,r1,j,r2,x,y}$ is 0 or τ_x is local to τ_i. The third constraint sets $\Pi_{i,r1,j,r2,x}$ to be 0 if τ_x is local to the writer of r_1.

Finally, $\Pi_{i,r1,j,r2}$ is set to 1 if there is a task τ_x (other than τ_i) accessing r_2 that has $\Pi_{i,r1,j,r2,x} = 1$. This can be enforced with the following constraint:

$$\begin{aligned} &\forall r_1 \neq r_2, \forall \tau_i \wedge \tau_i \in rw(r_1), \forall \tau_j \neq \tau_i \wedge \tau_j \in rw(r_2) : \\ &\Pi_{i,r1,j,r2} = \max_{\tau_x \neq \tau_i, \tau_x \in rw(r_1)} \Pi_{i,r1,j,r2,x}. \end{aligned} \tag{3.29}$$

$\Pi_{i,r1,j,r2}$ is 1 if r_2's access does not affect the response time of $C'_{i,(r1)}$. Thus, the special case of equal ceilings between $C'_{i,(r1)}$ and $C'_{j,(r2)}$ also leads to $\Pi_{i,r1,j,r2} = 1$. This situation arises only if the highest priority remote tasks for both r_1 and r_2 are the same. For example, if τ_1 accesses resource r_9 in Fig. 3.1. By setting $\pi_{i,i} = 1$ for all τ_i, this scenario can be incorporated into the existing analysis.

Remote blocking time BR_i

The remote blocking time in MPCP is calculated according to Eqs. (2.6)–(2.8). The remote blocking BR_i for τ_i is the sum over all resources accessed by τ_i (Eq. (2.8)), represented in the following constraint:

$$\forall \tau_i : \; BR_i = \sum_{r:\tau_i \in rw(r)} BR_{i,r}. \tag{3.30}$$

Accesses to global resources in MPCP are queued by priority. When a task tries to access a global resource, it can be blocked by (a) a lower priority task holding the resource (first term in Eq. (2.7)); (b) higher priority tasks holding the resource or queued waiting for it (second term in Eq. (2.7)). At most one

lower priority task can block τ_i. A higher priority task τ_k can block τ_i multiple times as shown in Eq. (2.7). We define two variables: $BR1_{i,r,k}$ ($BR2_{i,r,k}$) for the remote blocking suffered by τ_i when it tries to access r and gets blocked by a lower (higher) priority task τ_k. With these definitions, $BR_{i,r}$ is:

$$\begin{aligned} &\forall \tau_i, \forall r \wedge \tau_i \in rw(r): \\ BR_{i,r} = &\max_{\tau_k \neq \tau_i, \tau_k \in rw(r)} BR1_{i,r,k} + \sum_{\tau_h \neq \tau_i, \tau_h \in rw(r)} BR2_{i,r,h}. \end{aligned} \tag{3.31}$$

Eq. (3.31) is equivalent to Eq. (2.7) with $BR1_{i,r,k}$ representing $W'_{k,(r)}$ and $BR2_{i,r,h}$ denoting $\left(\left\lceil \frac{B^r_{i,j}}{T_h} \right\rceil + 1\right) W'_{h,(r)}$. $BR1_{i,r,k}$ is computed according to Eq. (2.7). When τ_k holds r, it can be preempted on its core by a task τ_a to execute a resource with a higher priority ceiling. $BR1_{i,r,k}$ is computed for each preempting task τ_a as follows:

$$\begin{aligned} &\forall \tau_i, \forall r \wedge \tau_i \in rw(r), \forall \tau_k \neq \tau_i \wedge \tau_k \in rw(r): \\ BR1_{i,r,k} \geq\ & C'_{k,(r)} + \sum_{\tau_a} BR_{i,r,k,a} - M(1 - \pi_{i,k}) - M \cdot W_r \\ & - M|wt(r) - i| \cdot G_{i,wt(r)} - M|wt(r) - k| \cdot G_{k,wt(r)}. \end{aligned}$$

The constraint ensures that (a) π_i has higher priority; (b) r is not protected by a wait-free mechanism; (c) the global resource r is accessed by τ_i; and (d) r is accessed by τ_k.

$BR1_{i,r,k,a}$ is the maximum blocking suffered by the remote critical section $C'_{k,(r)}$ due to the local task τ_a executing a remote section with a higher ceiling,

hence:

$$\begin{aligned} &\forall \tau_i, \forall r, \wedge \tau_i \in rw(r), \forall \tau_k \neq \tau_i \wedge \tau_k \in rw(r), \\ &\forall r_2 \neq r, \forall \tau_a \neq \tau_k \wedge \tau_a \in rw(r2) : \\ BR1_{i,r,k,a} \geq\ & C'_{a,(r2)} - M \cdot \Pi_{k,r,a,r2} - M(1 - G_{k,a}) \\ & -M|wt(r_2) - a| \cdot G_{a,wt(r2)} - M \cdot W_{r2}. \end{aligned} \tag{3.32}$$

This constraint ensures that (a) only critical sections with higher priority ceilings ($\Pi_{k,r,a,r2} = 0$) are accounted for; (b) τ_a is local to τ_k; (c) τ_a accesses the global resource r_2; and (d) the resource r_2 is not protected by a wait-free mechanism.

The calculation of $BR2_{i,r,k}$ proceeds similarly. However, the term $\left(\left\lceil \frac{B^r_{i,j}}{T_h} \right\rceil + 1\right)$ must be accounted for. Similarly to the overlap variables (OL, $OL2$, OR), we define OB as:

$$\forall \tau_i \neq \tau_h, \forall r \wedge \tau_i \in rw(r) : 1 + \frac{BR_{i,r}}{T_h} \leq OB_{i,r,h} < 2 + \frac{BR_{i,r}}{T_h}.$$

The calculation of $BR2_{i,r,k}$ then proceeds as follows:

$$\begin{aligned} &\forall \tau_i, \forall r \wedge \tau_i \in rw(r), \forall \tau_k \neq \tau_i \wedge \tau_k \in rw(r) : \\ BR2_{i,r,k} \geq\ & OB_{i,r,k} \cdot C'_{k,(r)} + \sum_{\tau_a} BR2_{i,r,k,a} - M \cdot \pi_{i,k} \\ & -M|wt(r) - i| \cdot G_{i,wt(r)} - M|wt(r) - k| \cdot G_{k,wt(r)} - M \cdot W_r; \end{aligned}$$

$$\begin{aligned} &\forall \tau_i, \forall r \wedge \tau_i \in rw(r), \forall \tau_k \neq \tau_i \wedge \tau_k \in rw(r), \\ &\forall r_2 \neq r, \forall \tau_a \neq \tau_k \wedge \tau_a \in rw(r_2) : \\ BR2_{i,r,k,a} \geq\ & OB_{i,r1,k} \cdot C'_{a,(r2)} - M \cdot \Pi_{k,r,a,r2} - M \cdot W_{r2} \\ & -M(1 - G_{k,a}) - M|wt(r_2) - a| \cdot G_{a,wt(r2)}. \end{aligned}$$

3.3.4 Objective function

For each wait-free resource, the memory cost is calculated using the temporal concurrency control protocol as in [40]. A variable CT_r represents the memory cost of resource r:

$$\begin{aligned} \forall r, \forall \tau_i \in rd(r), \quad CT_r \geq (1 + \max(2, 1 + \lceil \frac{T_i}{T_{wt(r)}} \rceil))D_r \\ -M \cdot G_{i,wt(r)} - M(1 - W_r), \end{aligned} \tag{3.33}$$

where D_r is the size of the shared data buffer r. The first term on the right side of the constraint is a constant and the last two terms ensure that the resource is global and protected by a wait-free mechanism. The objective is to minimize the total memory overhead:

$$\min \sum_{r \in \mathcal{R}} CT_r. \tag{3.34}$$

3.4 Heuristic Algorithms

While the MILP formulation presented in the previous section can find optimal system configurations (a feasible solution if one exists with minimum memory cost), the problem becomes too complex for large designs. To address this scalability issue, we present two heuristic algorithms. The first is a wait-free extension of the GS algorithm that uses wait-free methods as a last resort when GS fails. While this extension can improve schedulability, it ignores memory costs when making assignment decisions. To optimize both schedulability and memory cost, we propose an entirely new memory-aware partitioning heuristic that searches a wider design space by initially assuming

all resources use wait-free methods. Memory cost is then reduced by using lock-based mechanisms whenever feasible.

3.4.1 Extending greedy slacker with wait-free methods

The first algorithm (GS-WF) extends the GS [90] heuristic. Once a task fails to be assigned to any core in GS, GS-WF uses wait-free mechanisms for all the global resources accessed by the task and attempts to assign the task to each of the cores. If this makes the task schedulable on at least one core, GS-WF selects the one that maximizes the smallest task slack normalized to the period; otherwise, the algorithm fails. GS-WF only tries to improve the assignments from GS in case of their failure. Given that the allocation decisions are inherently based on minimizing slack and do not consider the memory cost, the end result can be schedulable but quite inefficient in terms of memory.

3.4.2 Memory-aware partitioning algorithm

The second heuristic, Memory-aware Partitioning Algorithm (MPA), consists of two phases:

1. Finding an initial feasible solution;

2. Improving the solution by exploring other possible solutions using a local search.

In the first phase, the algorithm focuses on obtaining an initial *schedulable* solution. To maximize the probability of finding such a solution, all global

resources are initially protected by wait-free methods. Local resources are managed using the Priority Ceiling Protocol (PCP) [35] or the Stack Resource Policy (SRP) [37] as they only introduce minimal blocking. The second phase reduces the memory cost resulting from the use of wait-free methods through local search to selectively change the data consistency mechanism to MPCP or MSRP. We first describe the concept of assignment urgency which we utilize in the algorithm.

Assignment urgency

In the proposed algorithm, tasks are allocated to cores one by one. The order has a significant impact on schedulability and memory cost. We propose the concept of *Assignment Urgency (AU)*, an estimate of the penalty (in schedulability or memory) if a task is not the next to be assigned. The assignment urgency AU_i of task τ_i is defined as:

$$AU_i = \begin{cases} M_{max} & \tau_i \text{ schedulable on 1 core} \\ \left| \min_j MC(\tau_i, p_j) - \min_{k \neq j} MC(\tau_i, p_k) \right| & \tau_i \text{ schedulable on } > 1 \text{ cores} \end{cases}$$

where $MC(\tau_i, p_j)$ is the memory cost of assigning τ_i to p_j using wait-free for all its global resources, and M_{max} is a value that is higher than the worst case memory cost of assigning any task to any core in the system:

$$M_{max} = \max_{\forall \tau_i \in \mathcal{T}} \max_{\forall p_j \in \mathcal{P}} MC(\tau_i, p_j) + 1. \tag{3.35}$$

The definition of M_{max} ensures that tasks schedulable on only one core will have higher AU values than those with more assignment options. If a task can be scheduled on more than one core, then there is usually enough allocation freedom to defer its assignment. This may come with a cost in memory since the task may have to be scheduled on a suboptimal core (if the most suitable core in terms of memory is not available). The possible penalty is estimated as the absolute difference in memory between the core resulting in the lowest memory needs, and the second best.

The main algorithm for the task allocation and resource protection selection is shown in Algorithm 3.1. It takes as input the task set $\mathcal{T}$ and the locking protocol (MPCP or MSRP) to be used for lock-based resources. The algorithm then works in two phases.

Phase 1

The first phase is a greedy procedure that assumes the use of wait-free methods for all global resources and tries to assign each task to the core on which it has the least memory cost. The function `TaskSort()` computes the assignment urgencies of a given list of (unassigned) tasks and sorts them by decreasing AU. It returns *failure* if a task has no feasible core. Algorithm 3.1 uses `TaskSort()` to sort the set of unassigned tasks in the list $\mathcal{LT}$ (Line 5). The task with the highest assignment urgency is then assigned to its best candidate core (Lines 13–14). At any time, if a task has no feasible core, the algorithm resets the task assignments and tries a resource-oblivious any-fit policy using the function `AnyFit()` (Lines 6–11). `AnyFit()` runs in the

Algorithm 3.1: Memory-aware Partitioning Algorithm

```
1: Function AllocationAndSynthesis(𝒯, lockProtocol)
2: Phase 1:
3: ℒ𝒯 ← 𝒯
4: while ℒ𝒯 ≠ ∅ do
5:     if TaskSort(ℒ𝒯) = failure then
6:         Reset partitioning
7:         if AnyFit() = true then
8:             Goto Phase 2
9:         else
10:            return failure
11:        end if
12:    else
13:        τ_k ← ExtractFirst(ℒ𝒯)
14:        TA ← TA+ allocating τ_k to p_j with smallest MC(τ_k, p_j);
15:    end if
16: end while
17:
18: Phase 2:
19: OptimizeResources(TA, lockProtocol)
20: curOpt ← TA; 𝒩𝒜.add(TA)
21: while 𝒩𝒜 ≠ ∅ do
22:    Th ← MemoryCost(GetLast(𝒩𝒜))
23:    curSys = ExtractFirst(𝒩𝒜)
24:    ℒ𝒩 ←GenerateNeighbors(curSys)
25:    for all AS in ℒ𝒩 do
26:        OptimizeResources(AS, lockProtocol)
27:        if IsSchedulable(AS) and MemoryCost(AS) < Th and NotVisited(AS) then
28:            𝒩𝒜.add(AS)
29:            if size(𝒩𝒜) > n then RemoveLast(𝒩𝒜)
30:            if cost(AS) < cost(curOpt) then curOpt = AS
31:            Th ← MemoryCost(GetLast(𝒩𝒜))
32:        end if
33:    end for
34:    if NoChange(curOpt, #iter) or cost(curOpt) ≤ tgtCost then
35:        return curOpt
36:    end if
37: end while
```

following order: worst-fit with decreasing utilization, best-fit with decreasing utilization, first-fit, and next-fit. If `AnyFit()` also fails, Algorithm 3.1 returns failure. Otherwise, at the end of the first phase, all tasks should be assigned to a core in a task allocation scheme TA.

Phase 2

In the second phase, the solution TA obtained in the first phase is improved with respect to memory cost by exploring selected neighbors (with smaller

memory cost). The **main loop** of phase 2 (Lines 21–37 of Algorithm 3.1) has some similarities with branch-and-bound algorithms. It explores the neighboring solutions among a controlled number of candidates, which are placed in a list $\mathcal{NA}$ sorted by increasing memory cost. The first solution (with lowest cost) in $\mathcal{NA}$ is further explored by branching (generating its neighbors).

The difficulty in exploration arises from the estimate of the quality of the solutions that may be found under a given branch. The algorithm allows the exploration of solutions (neighbors) with both lower and higher costs than the current optimum to avoid getting stuck in a local optimum. However, to avoid infinite searches, the exploration is bounded by a condition on the number of iterations without improvement (Line 34, Algorithm 3.1). It is also bounded by the size of $\mathcal{NA}$ which is at most equal to the number n of tasks in the system (Line 29, Algorithm 3.1). Essentially, the best n unexplored candidates at any stage are kept in $\mathcal{NA}$. In our experiments, larger sizes for $\mathcal{NA}$ such as $2n$ or $4n$ do not improve the quality of the obtained solution but result in substantially longer runtimes. To avoid loops, recently visited neighbors are discarded.

The solution space exploration works as follows. The first solution in $\mathcal{NA}$ (solution with minimum cost) is removed from the list (Line 23) and considered as the new base ($curSys$) for further exploration. Lines 24–33 show the generation and exploration of neighbors. All feasible neighbors of the current base $curSys$ are generated (Line 24). However, not all of them are further explored. A threshold value Th is used as an acceptance criterion for new neighbors generated from $curSys$, which is set to be the memory cost of the last solution in $\mathcal{NA}$ (the unexplored solution with the n-th highest cost, as

in Line 31). Only solutions with lower costs than Th are accepted for further exploration (added to $\mathcal{NA}$, Lines 27–28). The cost of any new solution is considered only after resource optimization (Line 26) such as when comparing with Th (Line 27), or when comparing with the current optimum *curOpt* (Line 30). Resource optimization for a given solution is performed by the function `OptimizeResources()` (discussed below).

If there are no more promising solutions to explore ($\mathcal{NA}$ becomes empty), or the solution is not improving after a given number of iterations, or a solution with the desired quality (*tgtCost*, typically depending on the available RAM memory) is found, the algorithm terminates (Lines 34–36).

The generation of neighbors is done inside the function `GenerateNeighbors()`. A *neighbor* of a given task allocation solution can be obtained by a re-assignment of a task τ_i allocated on core p_a to a different core $p_b \neq p_a$ that can accommodate it, either directly (1-move neighbor) or by removing a task τ_j with equal or higher utilization from p_b and assigning τ_j to another core $p_c \neq p_b$ (2-move neighbor).

Resource optimization is performed by the function `OptimizeResources()`. This function changes the protection mechanism of global resources from wait-free to lock-based (MPCP or MSRP) for as many resources as possible, while retaining system schedulability. It is called at the start of phase 2 (Line 19, Algorithm 3.1) and whenever a new candidate solution is found (Line 26, Algorithm 3.1). An exhaustive search in `OptimizeResources()` would be impractical, therefore, we developed a heuristic (Algorithm 3.2) that initializes the protection mechanism for all the global resources to wait-free and places

Algorithm 3.2: Optimizing data consistency mechanisms

```
1: Function OptimizeResources(TA, lockProtocol)
2: GR = FindGlobalResources(TA)
3: for all r_i in GR do
4:     SetProtocol(r_i, wait_free)
5: end for
6: SortByMemoryCost(GR)
7: for all r_i in GR do
8:     SetProtocol(r_i, lockProtocol)
9:     if IsSchedulable(TA) = false then
10:        SetProtocol(r_i, wait_free)
11:    end if
12: end for
```

them in a list $\mathcal{GR}$ (Lines 2–5). It then sorts the resources in $\mathcal{GR}$ by decreasing memory cost of their wait-free implementation (Line 6). The protection mechanism of the first (highest cost) resource in $\mathcal{GR}$ is changed to a lock-based mechanism and system schedulability is checked. If the system becomes unschedulable, the protection mechanism is reverted to wait-free. The procedure iterates through all the resources in $\mathcal{GR}$, changing the protection mechanism from wait-free to lock-based whenever possible (Lines 7–12).

3.5 Experimental Results

In this section, the proposed approaches are evaluated in terms of schedulability and memory cost (if schedulable). In Section 3.5.1, we first compare the schedulability analysis from [17] (Eqs. (2.11)–(2.15)) and the ILP-based schedulability analysis in [38] using relatively small task sets. Then, we focus on the performance of the proposed heuristics and provide an overall evaluation of the heuristics in Section 3.5.2 comparing the algorithms presented in the previous sections with state-of-the-art partitioning algorithms. For MSRP, we compare GS-WF and MPA with the best performing resource-aware task

allocation algorithms proposed so far: GS and CASR. For MPCP, we compare our algorithms with SPA [14], BPA [91], GS applied to MPCP, and CASR applied to MPCP. In Section 3.5.3, we study the robustness of the algorithms under different parameter variations. Finally, in Section 3.5.4, we focus on the ILP formulations and compare them with heuristics that have the option to use wait-free resources (GS-WF, MPA). Through the experiments, the unit for memory cost is in bytes. For each parameter configuration, 100 systems are randomly generated.

We adopt a task generation scheme similar to [90]. We consider systems with 3, 4 or 8 cores. The task periods are generated according to a log-uniform distribution from one of two different ranges [10, 100] ms and [3, 33] ms. The average task utilization is selected from the set {0.05, 0.1, 0.12, 0.2, 0.3}. The critical section lengths are randomly generated in either [0.001, 0.1] ms or [0.001, 0.015] ms with uniform distribution.

The tasks share between 1 and 40 resources. The resource sharing factor (rsf) represents the portion of tasks in the system sharing a given resource, e.g., $rsf = 0.1$ means each resource is shared by 10% of the tasks in the system. For each experiment, a resource sharing factor is selected from the set {0.1, 0.25, 0.5, 0.75}. The tasks that share a given resource are randomly generated. This process is independent for each resource. The size of communication data is chosen randomly from the set (in bytes): 1 (with probability $p = 10\%$), 4 ($p = 20\%$), 24 ($p = 20\%$), 48 ($p = 10\%$), 128 ($p = 20\%$), 256 ($p = 10\%$), and 512 ($p = 10\%$).

It is well known that as the utilization of the task set increases, schedu-

lability decreases. To be able to quantitatively compare the schedulability of different approaches, we devise a new metric which we refer to as the *critical utilization*. In a given experiment, the critical utilization for a particular partitioning algorithm is the maximum utilization at which the partitioning algorithm is able to schedule at least 95% of generated systems. This characterizes the point where *almost* all systems are schedulable while leaving some margin in case a particular approach fails for a few systems.

As CASR is a tunable algorithm requiring setting the utilization bound (U_b) value, we evaluate two approaches: 1) a single run of CASR with utilization bound $U_b = U_{\mathcal{T}}/m$ (denoted by CASR or s-U_b); and 2) the best solution from multiple (five) runs of CASR with a set of U_b values $\{0, 25\%, 50\%, 75\%, 100\%\}$ (denoted by CASR-m or m-U_b). For MPA, the number of iterations is set to be $10n$ where n is the number of tasks in the system.

3.5.1 Schedulability analysis

We first focus on schedulability analysis and evaluate the task partitioning algorithms using the schedulability analysis for MSRP in [17], and the ILP-based analysis in [38]. The average core utilization is selected to be in the range 68%–96% by fixing the average task utilization at 12% and varying the number of tasks in the range [17, 24]. Utilizations outside this range show no variation between the approaches. Tasks are to be allocated to 3 cores, and share 20 resources. The resource sharing factor is selected to be 0.25. The task periods are randomly generated in [10, 100] ms and critical section lengths are randomly chosen in [0.001, 0.1] ms.

Table 3.3 reports the percentage of schedulable solutions obtained by each algorithm and the memory costs (for MPA and GS-WF). Table 3.4 presents the average runtime of each algorithm. Each table is divided into two parts where the upper half shows the results using the analysis in [17], and the lower half (with the prefix *i*) shows the results for the ILP-based analysis [38]. The general trend is that MPA outperforms the other algorithms. The performance of all partitioning algorithms improves in terms of both schedulability and memory cost when using the ILP-based analysis. However, this comes at the price of a much longer runtime. The runtimes for these relatively small systems increase by 150 times on average when using the ILP schedulability analysis, and even more for larger systems. In general, the *relative comparison* among the partitioning schemes does not appear to be sensitive to the choice of the analysis method. In the rest of the experiments, we use the analysis in [17] for MSRP and the analysis in [14, 36] for MPCP with larger systems.

Table 3.3: Schedulability/average memory cost (GS-WF and MPA only, in bytes)

Average core util	68%	80%	88%	92%	96%
GS	100%	46%	0%	0%	0%
CASR (s-U_b)	100%	69%	0%	0%	0%
CASR (m-U_b)	100%	91%	1%	0%	0%
GS-WF	100%/0	100%/1687	26%/6963	0%/–	0%/–
MPA	100%/0	100%/0	96%/231	37%/3315	0%/–
iGS	100%	99%	70%	9%	0%
iCASR (s-U_b)	100%	100%	85%	21%	0%
iCASR (m-U_b)	100%	100%	98%	37%	0%
iGS-WF	100%/0	100%/31	81%/347	13%/1395	0%/–
iMPA	100%/0	100%/0	96%/0	37%/79	0%/–

Table 3.4: Average runtime (in seconds)

Average core util	68%	80%	88%	92%	96%
GS	0.022	0.026	0.011	0.007	0.007
CASR (s-U_b)	0.016	0.044	0.035	0.002	0.001
CASR (m-U_b)	0.019	0.073	0.159	0.009	0.007
GS-WF	0.020	0.032	0.021	0.009	0.008
MPA	0.077	0.183	13.200	3.700	0.022
iGS	65.31	86.060	101.990	49.97	20.160
iCASR (s-U_b)	28.490	70.530	122.910	132.74	51.700
iCASR (m-U_b)	41.540	86.600	164.290	398.200	253.220
iGS-WF	47.310	88.670	104.650	57.830	26.130
iMPA	198.62	247.730	133.430	52.960	17.350

3.5.2 General evaluation of heuristics

In the second set of experiments, we focus on evaluating the approaches when applied to larger systems. For this experiment, the number of tasks n is varied in the range [40, 76], to be scheduled on 8 cores. The average utilization of each task is 0.1. Periods are generated in the range [10, 100] ms, and critical section lengths are selected in [0.001, 0.1] ms. The number of resources is fixed at 4, and each resource is shared by a quarter of the systems' tasks.

The results are shown in Figs. 3.2–3.5. Fig. 3.2 shows that when MPCP is used as the data consistency mechanism, SPA [14] and BPA [91] perform worst with respect to schedulability. BPA and SPA both try to divide the tasks in the system into bundles that share resources, and then, attempts are made to assign the bundles to processors. While this strategy might work for some applications, in general it is difficult to find bundles with suitable utilizations that only share resources internally. When these algorithms fail to find such bundles, they do not perform as well as the other algorithms. Even

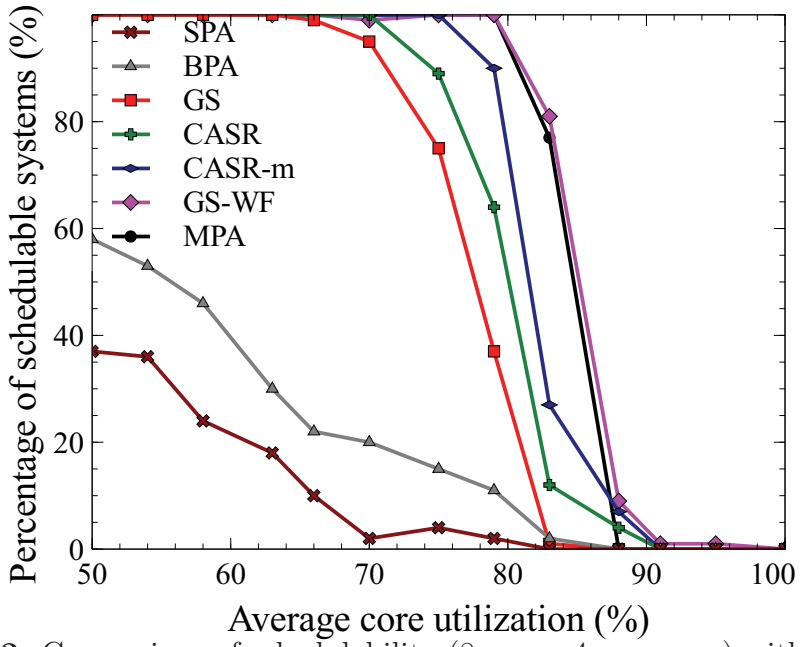


Fig. 3.2: Comparison of schedulability (8 cores, 4 resources) with MPCP

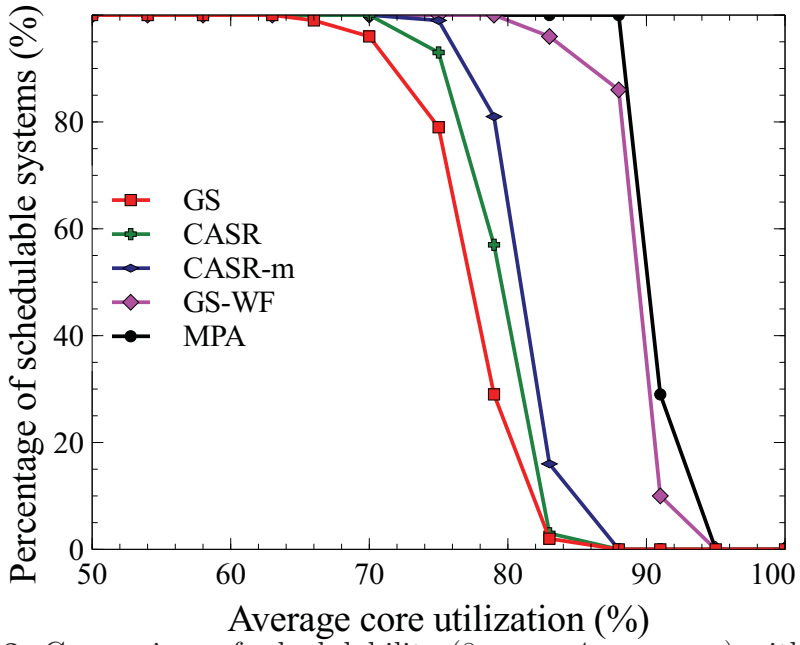


Fig. 3.3: Comparison of schedulability (8 cores, 4 resources) with MSRP

at utilizations as low as 50%, SPA can only schedule 37% of systems and BPA 58% of systems while all other algorithms easily schedule 100%

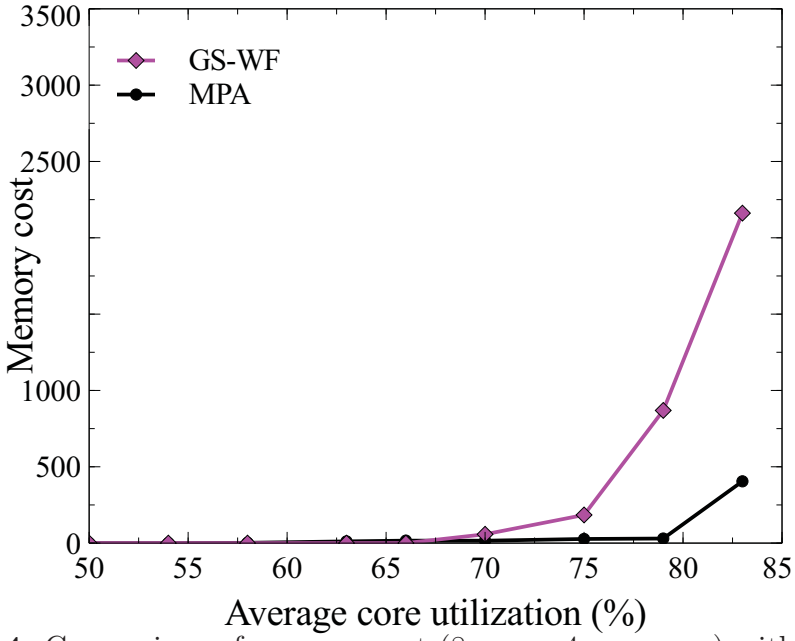


Fig. 3.4: Comparison of memory cost (8 cores, 4 resources) with MPCP

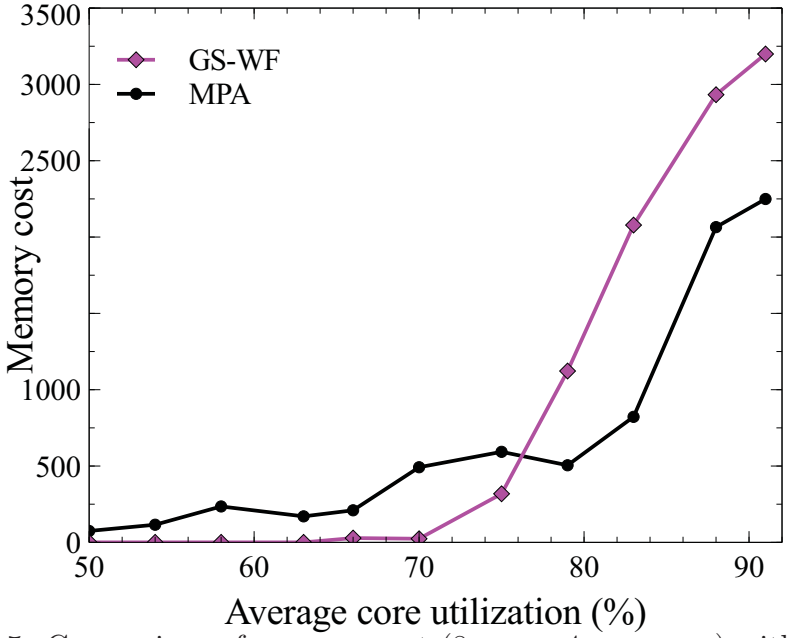


Fig. 3.5: Comparison of memory cost (8 cores, 4 resources) with MSRP

Figs. 3.2 and 3.3 illustrate that CASR outperforms other algorithms that do not use wait-free methods regardless of the blocking mechanism used (MPCP

or MSRP). The critical utilization (maximum utilization at which the partitioning algorithm is able to schedule 95% or more of generated systems) is improved from 70% for GS to 76%-77% for CASR-m. These two figures demonstrate the improvement obtained using wait-free methods. GS-WF and MPA provide significantly better schedulability than GS, increasing the critical utilization to 80% for both MPA and GS-WF with MPCP and to 83% for GS-WF and 88% for MPA with MSRP. Figs. 3.4 and 3.5 compare the additional memory cost needed by wait-free methods for both GS-WF and MPA. As shown in Fig. 3.4, MPA requires 15.4% of the memory required by GS-WF (56 bytes on average compared to 363 for GS-WF) with MPCP. For MSRP (Fig. 3.5), in systems with small utilization, GS-WF performs noticeably better. However, as the average core utilization exceeds 0.75 (number of tasks exceeds 60), MPA tends to have a lower (about 40% less on average) cost. These figures show that system schedulability can be improved with minimal memory penalty when lock-based resources are selectively replaced by wait-free ones. Algorithms (such as GS-WF and MPA) that exploit this observation outperform those that don't.

3.5.3 Effect of different parameters

In this section, we evaluate the robustness of the proposed heuristics to changes in the input parameter settings. To evaluate systems with more heavy communication, we first generate system configurations with 20 resources. The number of tasks is varied in the range [20, 40] to be scheduled on 4 cores with rsf=0.25. All other parameter settings are the same as those in Section 3.5.2.

Figs. 3.6–3.9 plot the results of this experiment. The general trend in schedulability remains the same as Section 3.5.2: CASR performs better than SPA, BPA, and GS. MPA significantly improves upon GS-WF in both schedulability and memory cost. In terms of schedulability; When using MPCP, the critical utilization is improved from 57% for GS to 65% for CASR-m, to 70% for GS-WF, to 76% for MPA. When using MSRP, the critical utilization is similarly improved from 66% for GS to 70% for CASR-m, to 78% for GS-WF, to 85% for MPA. The memory advantage for MPA over GS-WF is significantly higher. MPA requires only about 1% of the memory required for GS-WF (44 bytes to 4505 bytes on average with MPCP, 32 bytes to 2719 bytes with MSRP) and this advantage is consistent across all utilizations. With more resources in the system (and hence more options to apply waitfree methods), MPA significantly improves system schedulability. Compared to the best algorithm that does not use waitfree methods (CASR-m), the critical utilization is improved by 17% for MPCP and 21% for MSRP. These results shows that a significant improvement in schedulability is possible with MPA at a minimal memory cost. Using MPA, more systems can be scheduled on a specific platform without the need for costly increases to the processing power of the system.

Next, we observe the algorithms performance when varying other system parameters. First, we fix the utilization (at 70% by having $n = 28$) and increase the processor load by increasing the number of shared resources in the range [1, 40] while fixing rsf at 0.25. The schedulability results are shown in Figs. 3.10 and 3.11. For MPCP, MPA is the only algorithm to always find a schedulable solution. For MSRP, both GS-WF and MPA always schedule all

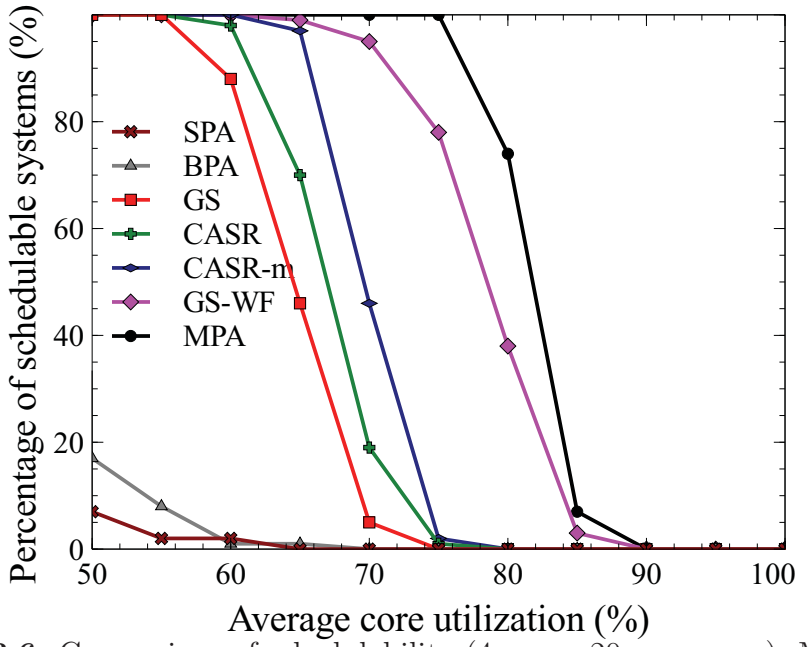


Fig. 3.6: Comparison of schedulability (4 cores, 20 resources)- MPCP

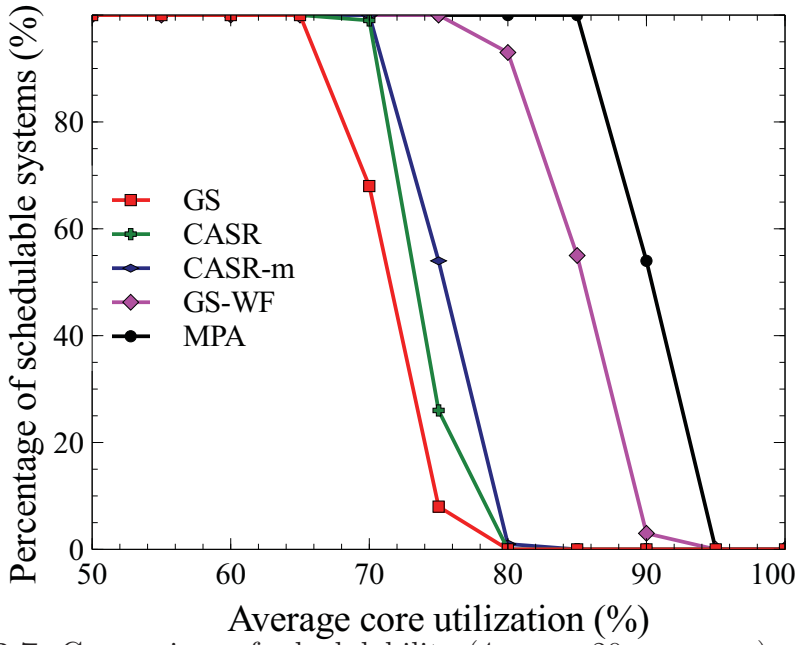


Fig. 3.7: Comparison of schedulability (4 cores, 20 resources) - MSRP

systems.

To evaluate the effect of changing the resource sharing factor, another ex-

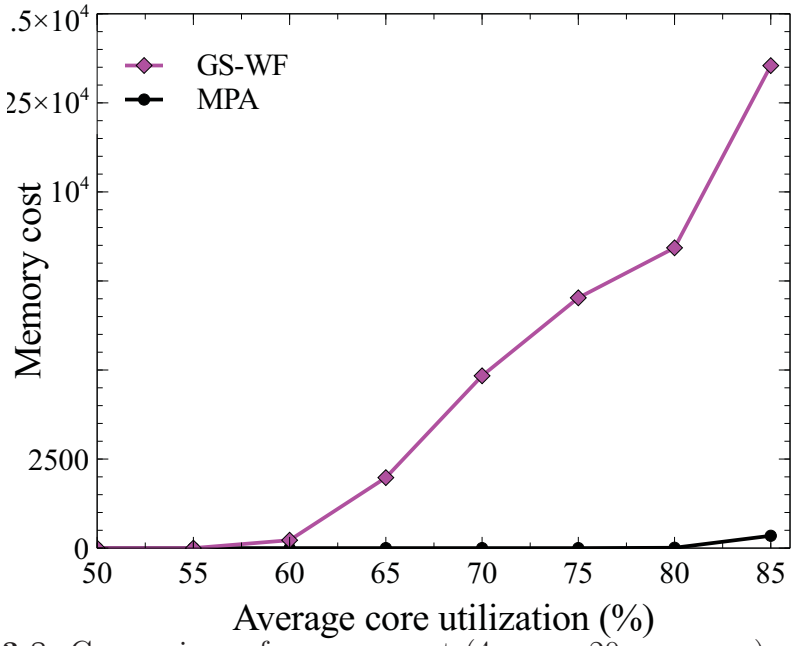


Fig. 3.8: Comparison of memory cost (4 cores, 20 resources) - MPCP

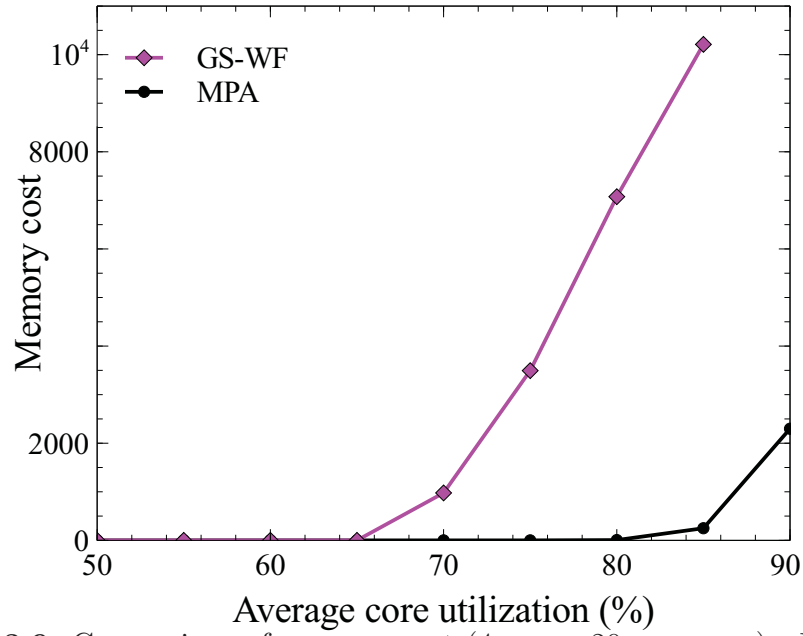


Fig. 3.9: Comparison of memory cost (4 cores, 20 resources) - MSRP

periment is performed using the same settings while keeping the number of resources at 20. Tables 3.5 and 3.6 present the result. It shows that MPA and

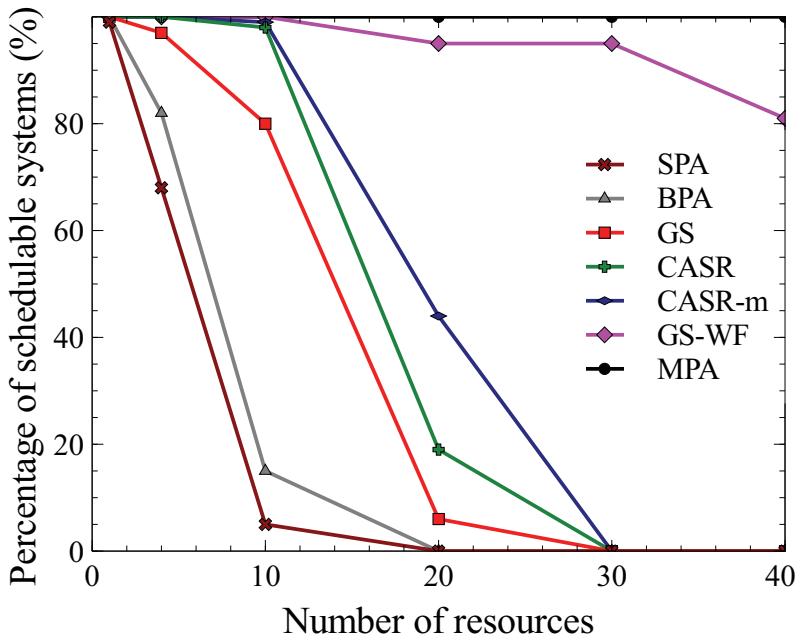


Fig. 3.10: Comparison of schedulability with a variable number of resources - MPCP

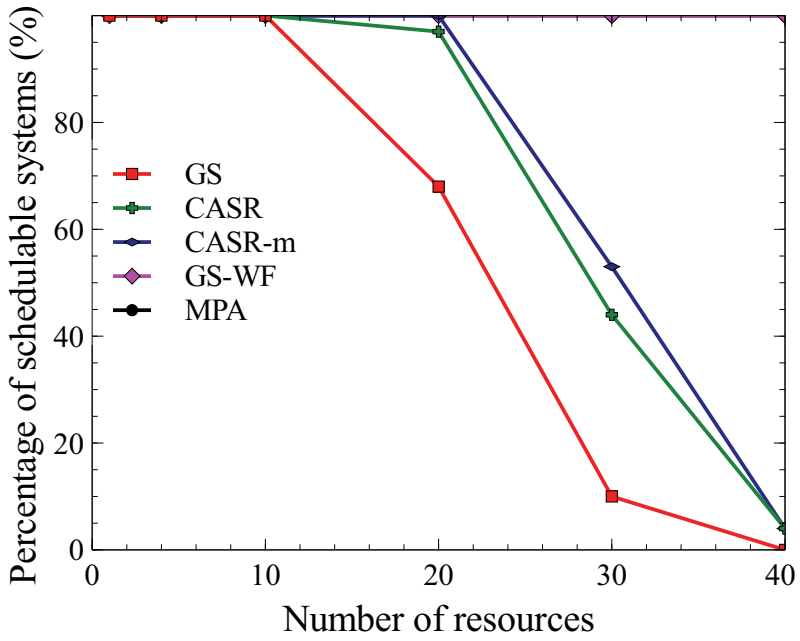


Fig. 3.11: Comparison of schedulability with a variable number of resources - MSRP

Table 3.5: MPCP schedulability/average memory cost (GS-WF and MPA only, in bytes) for different sharing factors

rsf	0.1	0.25	0.5	0.75
SPA	52%	0%	0%	0%
BPA	73%	0%	0%	0%
GS	98%	8%	0%	0%
CASR (s-U_b)	100%	17%	0%	0%
CASR (m-U_b)	100%	38%	0%	0%
GS-WF	99%/1.45	96%/4731	81%/10145	25%/12523
MPA	100%/0	100%/0	100%/19	96%/818

GS-WF perform significantly better than all other algorithms as tasks communicate more often. At lower rsf values, there is little room for improvement for MPA over GS-WF since it is possible to schedule 100% of systems however, with more sharing a more significant improvement for MPA is observed. MPA also succeeds in keeping the memory usage small. For example, when $rsf = 0.75$ MPA requires on average just 818 bytes of memory with MPCP and 841 bytes of memory with MSRP, compared to GS-WF's 12523 and 11493 (a reduction of 93.5% and 92.7% respectively). Experiments varying other parameters, e.g., periods and critical section lengths, yield similar conclusions. The results in this section show that the underlying premise that selectively replacing lock-based resources with waitfree ones improves schedulability holds when system parameters are changed and that our proposed MPA heuristic exploits this observation and is robust to parameter changes.

Table 3.6: MSRP schedulability/average memory cost (GS-WF and MPA only, in bytes) for different sharing factors

rsf	0.1	0.25	0.5	0.75
GS	100%	68%	0%	0%
CASR (s-U_b)	100%	94%	0%	0%
CASR (m-U_b)	100%	100%	0%	0%
GS-WF	100%/0	100%/1136	98%/9628	67%/11493
MPA	100%/0	100%/0	100%/104	100%/841

3.5.4 Evaluation with ILP

Finally, we implemented the ILP formulation presented in Section 3.3 using CPLEX [92], and compared it with the two heuristics that have the option of using wait-free methods (MPA and GS-WF). A 4-hour time out was used, after which CPLEX returned the best solution found so far (if any). We generated smaller systems consisting of 15 tasks sharing 8 resources. The tasks were scheduled on 3 cores, with each resource being shared by 4 tasks. We observed schedulability and memory cost as utilization was increased in the range [0.5-0.9]. Other parameters were similar to the previous experiments. The results are summarized in Tables 3.7 (for MPCP) and 3.8 (for MSRP). These results show that with a 4-hour timeout, the ILP formulation manages to schedule more systems than the proposed heuristics. The ILP achieves a lower memory cost than GS-WF. However, it does not always achieve a lower memory cost than MPA, especially for MPCP. This is due to the timeout used which forces the MILP to exit with a sub-optimal solution when it can not find an optimal one quickly. Setting a larger timeout value shall allow the ILP to find solutions with reduced memory cost.

As the task size grows larger, it becomes difficult for the ILP to find schedulable systems with low cost within a reasonable time. We performed an experiment with systems consisting of 20 tasks where the MSRP ILP formulation performed worse in schedulability than MPA with a 2-hour timeout. Increasing the timeout to 12 hours, ILP just edged out MPA in schedulability. The MPCP formulation was even more difficult for the ILP solver to handle, requiring more solver time to find schedulable systems. This scalability issue is expected since the problem is NP-hard. The approaches proposed in this chapter complement each other: the ILP formulation can be used to find optimal solutions for smaller task sets, while the MPA algorithm can be used to find sub-optimal ones for larger task sets.

Table 3.7: MPCP Schedulability/average memory cost (in bytes) at different utilizations

Avg. core util	50%	60%	70%	80%	90%
GS-WF	100%/0	100%/0	100%/53.48	83%/918.1	3%/3205
MPA	100%/0	100%/0	100%/0	96%/2.78	3%/384
ILP (4h)	100%/0	100%/0	100%/0.03	100%/32.78	27%/939

Table 3.8: MSRP Schedulability/average memory cost (in bytes) at different utilizations

Avg. core util	50%	60%	70%	80%	90%
GS-WF	100%/0	100%/0	100%/0	100%/145.9	39%/1749
MPA	100%/0	100%/0	100%/0	100%/0	72%/104.4
ILP (4h)	100%/0	100%/0	100%/0	100%/0	96%/99.68

3.6 Conclusion

In this chapter, we proposed new resource-aware partitioning approaches for fixed priority partitioned multicore systems. Due to the complexity of the problem, one solution approach might not be scale to all systems. Therefore, we propose two approaches; First, an ILP formulation is proposed to perform task allocation, priority assignment, and protection mechanism selection between lock-based and wait-free methods, to find a schedulable system with minimal memory cost. Second, for large systems where ILP cannot scale to, we proposed the MPA heuristic. Experimental results show that MPA improves the critical utilization (maximum utilization at which at least 95% of systems are found schedulable) to the 76%-88% range allowing designers to schedule more systems on the same hardware with minimal memory cost. Experimental results also show that this improvement is robust to changes in the input parameter settings.

Chapter 4

Task Allocation in Mixed-Criticality Systems

MCS have gained increasing interest in the past few years due to their industrial relevance. We focus in this chapter on finding an efficient scheduling approach for MCS on multicore architectures. When MCS are implemented on a multicore platform, previous work indicates that partitioned scheduling is superior to global scheduling [48]. However, partitioned scheduling often comes with a cost in the overall utilization achievable since some processors will have unused capacity. This problem is even worse for mixed-criticality tasks, as tasks have different execution times (and hence utilizations), at different criticality levels. A partitioning suitable for one criticality level might not be efficient at another criticality level. We propose in this chapter[1], to use

[1]The work in this chapter was done in collaboration with Qingling Zhao and Ahmed Youssef. My contribution was proposing the use of the dual partitioned approach, and the main development and implementation of the DPM algorithm with input from the other authors. The scheduling analysis was developed by Qingling Zhao.

a semi-partitioned scheduling approach, which we refer to as *dual-partitioned scheduling*, to schedule multicore MCS.

In dual-partitioned scheduling, HI-criticality tasks are statically mapped to processors at all times to ensure predictability while LO-criticality tasks are allowed, with limited migration, to efficiently use the cores. As long as the system remains in a stable criticality mode, it is fully partitioned. However, during criticality change, LO-criticality tasks can be migrated to another core. In this chapter, we present the Dual Partitioned Mixed-Criticality (DPM) algorithm for performing task allocation in MCS. Our experiments show that DMP performs consistently better than the best full partitioned algorithms across a wide range of parameter settings. For example, at utilizations of 0.8 or higher, the approach is able to enhance the schedulability of full partitioned algorithms by 17%.

The rest of this chapter is organized as follows: In Section 4.1, the system model is presented. The proposed dual-partitioned mixed-criticality scheduling approach and the schedulability analysis are described in Section 4.2. Section 4.3 presents the experimental results. Finally, Section 4.4 concludes the chapter.

4.1 System Model

4.1.1 Elastic Mixed-Criticality (EMC)

In the Adaptive Mixed-Criticality (AMC) model (Section 2.4), all LO-criticality tasks are abandoned in the HI-mode. While this guarantees sufficient CPU time for HI-criticality tasks, no guarantees are provided for LO-criticality

tasks in the HI-mode which might be undesirable for many applications. For example, in control applications, sporadic delays can be introduced and cause system instability [93]. To relax the assumptions of AMC and provide a level of service to LO-criticality tasks, Burns et al. [45] propose increasing the period of the LO-criticality tasks in HI-mode, a concept employed previously in the elastic task model [94].

Su et al. consider the elastic mixed-criticality task model, and study Earliest Deadline First (EDF) based scheduling algorithms for uniprocessors [95] and multicore processors [96]. [97] present the *Extended Elastic Mixed-Criticality (E^2MC)* task model, which assigns each LO-critical task a pair of small and large periods, representing its Quality of Service (QoS) guarantees in LO and HI-crit modes, respectively, and presents a schedulability test for a mode-switch EDF scheduler based on unified Demand Bound Analysis.

[98, 99] present ZS-QRAM, a scheduling approach that enables the use of flexible execution times and *application utility* to tasks in order to maximize total system utility of MCS. ZS-QRAM allows task periods to vary at runtime, with higher utility values associated with smaller periods. Both ZS-QRAM and EMC are motivated by the elastic task model [94], but their underlying task models are very different. ZSRM and ZS-QRAM are designed for maximizing the total system utility of soft real-time systems, while AMC and EMC are designed to achieve safety certification at multiple criticality levels.

In this chapter, we use the elastic task model by allowing LO-criticality tasks to run in the HI-mode at a reduced rate. We focus on fixed priority systems, hence, the elastic model is applied to fixed priority scheduling, instead

of EDF scheduling as in [95, 96].

A recently published work ([100]) also studies fixed priority scheduling with the elastic task model and introduces the Mode-Switch Fixed-Priority (MS-FP) scheduler. [100] focuses on priority and period selection to optimize control performance on a single core architecture while the focus of this chapter is on task mapping in multicore architectures.

4.1.2 Task model

We consider a system, consisting of a set of N independent mixed-criticality sporadic tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$, and M processors $P = \{\pi_1, \pi_2, ..., \pi_M\}$. In accordance with the current literature on MCS [12], we assume that (a) tasks are independent (i.e., there is no blocking due to shared resources), (b) tasks do not suspend themselves, other than at the end of their computation, (c) the overheads due to context switching, migration, etc., are negligible (assumed to be zero), (d) tasks have implicit deadlines equal to their periods (i.e., $D_i = T_i$).

Each task τ_i has a 5-tuple of parameters $\langle L_i, C_i(LO), C_i(HI), T_i(LO), T_i(HI)\rangle$ where

- $L_i \; \epsilon$ {LO, HI} denotes the task's criticality level.
- $C_i(LO)$ denotes the task's worst-case execution time (WCET) in LO-mode.
- $C_i(HI)$ denotes the task's WCET in HI-mode, with $C_i(HI) = C_i(LO)$ if $L_i = LO$, and $C_i(LO) \leq C_i(HI)$ if $L_i = HI$. $Cfactor$ denotes the ratio of the HI-to-LO WCET of the task ($Cfactor_i = C_i(HI)/C_i(LO)$).

- $T_i(LO)$ denotes the period (minimum arrival interval) of τ_i in LO-mode.
- $T_i(HI)$ denotes the period of τ_i in HI-mode,with $T_i(HI) = T_i(LO)$ if $L_i = HI$, and $T_i(HI) \geq T_i(LO)$ if $L_i = LO$.

LO-criticality tasks are guaranteed a reduced level of service ($T_i(HI) \geq T_i(LO)$) in the HI mode. For HI-criticality tasks, a more conservative assumption for the WCET is used in the HI-mode. Each task τ_i can thus have two different utilizations $(u_i(LO), u_i(HI))$ and consequently the system will have two different utilizations $(U(LO), U(HI))$.

We assume that Rate-Monotonic scheduling is used to schedule tasks on each processor. Since the periods of LO-criticality tasks are allowed to change in the HI-mode, all tasks are allowed to have two priorities: one for the LO-mode $pr_i(LO)$, and one for the HI-mode $pr_i(HI)$.

4.1.3 System behavior

Basic system operation proceeds mostly as described in Section 2.4.1. However, some changes are necessary since we are assuming that LO-criticality tasks must be guaranteed a reduced level of service ($T_i(HI) \geq T_i(LO)$) in the stable HI mode. The following change is made to step 4 in the system operation description in Section 2.4.1:

4- When this happens (a task executes beyond $C(LO)$), LO tasks are dropped immediately for *the duration of the mode change period only.* HI tasks continue execution and the HI task with the highest priority executes on the processor. The stable mode commences once all tasks are migrated and all HI-criticality carry-over jobs (those that started before the mode change)

have finished execution. At that point, LO tasks are allowed to continue executing again on their (possibly) new processors with reduced periods. HI tasks continue to execute through the entire mode change and in the HI mode up to their certification authority accepted WCET estimate (C(HI)).

We assume all processors go into HI-criticality mode at the same time. To the best of our knowledge, this is consistent with previous works on MCS in multicore platforms. We made the assumption that all carry-over jobs from LO-criticality tasks are dropped during a mode change. This is a reasonable assumption since we no longer need to guarantee them a maximum service level when a system goes into HI-mode. Migration can be implemented as a background task once mode change starts.

4.2 Dual-Partitioned Mixed-Criticality Scheduling

In this work, we focus on the issue of efficient scheduling of fixed priority tasks in MCS on a multicore architecture. MCS behave differently based on the mode of the system. As can be seen from the task model in Section 4.1, the properties of a given task differ from one mode to another. If the system is fully partitioned, this leads to a change in processors' utilizations in different modes. The change in processors' utilizations is not necessarily uniform across all processors since it depends on the properties of the tasks allocated to the processor and the method used by the certification agency to derive their WCET. The certification authority might determine, for example, that the WCET of a certain HI-criticality task might need to be increased by a large factor while another task's WCET does not need to be adjusted at all.

Therefore, the free capacity available to LO-criticality tasks across processors does not necessarily change by the same proportion when a mode change occurs. Currently, partitioning in MCS is done such that the requirements of HI-criticality tasks in the HI-mode and the requirements of the LO-criticality tasks in the LO-mode are both satisfied at the same time. This can be overly pessimistic.

The problem of assigning tasks to processors is a general case of the NP-hard bin-packing problem. Several bin-packing heuristics such as Best-Fit, Worst-Fit, and First-Fit have been applied to multiprocessor scheduling of traditional task sets [10] and recently used for MCS [52].

To enhance the efficiency of these partitioning algorithms when applied to MCS, we propose to use **dual-partitioned** mixed-criticality scheduling. By dual-partitioned we mean that the system in the steady modes (LO-mode and HI-mode) is fully-partitioned. However, the task-to-processor assignments in these two modes are not necessarily the same. A LO-criticality task τ_i in the system may have two (possibly different) designated processors $\pi_i(LO)$ and $\pi_i(HI)$. This approach avoids the shortcomings of global scheduling, in particular the need to use the pessimistic bounds of global scheduling, while being capable of scheduling more systems than partitioned scheduling, as shown by the experimental results in Section 4.3.

In terms of system operation, while the system continues to operate in a given mode, all tasks are executed on their designated processors. However, when a mode change occurs, tasks may migrate to a new set of designated processors. The set of tasks that migrate is determined by our algorithm at

design time. Tasks remain on the new processors while the system remains in the new mode. Migration is done to ensure efficient use of resources in the new mode. To preserve the predictability of critical tasks, we do not allow HI-criticality tasks to migrate. Only LO-criticality tasks can migrate such that they are guaranteed a minimum service level on the new processors.

To illustrate the principle of dual-partitioned MCS, we use a simple system consisting of 4 tasks to be partitioned onto two processors $\{P_1,P_2\}$ as shown in Table 4.1. We assume that deadlines are equal to periods and that rate-monotonic scheduling is used. In the case of tasks with equal periods, the task with the lower task ID has higher priority.

Without dual partitioned scheduling, it is impossible to schedule the system as any task allocation will result in a system utilization that exceeds 1 in one of the two modes. In the HI mode, task τ_1 has a utilization of 0.9, which means it cannot be co-located to the same core with any other task. However, in the LO-mode the total utilization for the other three tasks would be 0.3+0.5+0.3=1.1 and hence it is not schedulable.

With dual partitioned scheduling, it is possible to have a schedulable system: In LO-mode, tasks τ_1 and τ_2 are assigned to core P_1, while τ_3 and τ_4 to P_2. In HI-mode, τ_2 migrates to P_2, and the other tasks remain on their original cores. Figure 6.2 shows the runtime behavior of this assignment. The system starts in LO mode, then at time $t = 13$, τ_1 executes for more than its $C(LO)$ and the system goes through a mode change into HI-mode. At this instant, τ_2 starts to migrate to P_2. At the same time, all active LO-criticality tasks (τ_3 in this case) are dropped and no new instances are allowed to run for

the whole mode change period. At $t = 19$, all carry-over jobs finish and the system starts to operate in the HI-mode with τ_2 running on P_2.

Table 4.1: Task Parameters for The Example System

Task	Criticality	C(LO)	C(HI)	T(LO)	T(HI)
τ_1	HI	3	9	10	10
τ_2	LO	3	3	10	20
τ_3	LO	5	5	10	20
τ_4	HI	3	6	10	10

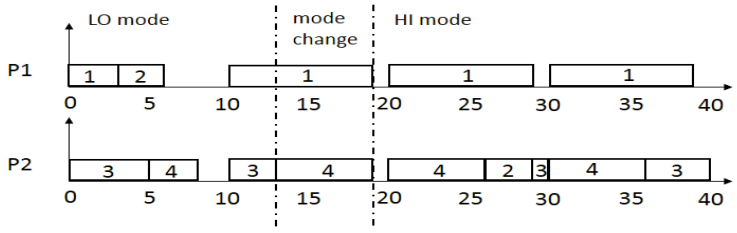


Fig. 4.1: Example system execution trace

The *Dual-Partitioned Mixed-Criticality* (DPM) algorithm shown in Algorithm 4.1 is proposed to perform task assignments. Two partitions are eventually produced by the algorithm: LO-partition is used for the LO-mode and HI-partition for the HI-mode. The algorithm can be viewed as consisting of two phases: the partitioning phase and the optimization phase.

First phase - Partitioning: Two partitions are produced, LO-partition and HI-partition. This is done in three steps: first we assign the HI-criticality tasks (same assignment in both modes), followed by LO-criticality tasks in the HI mode, and finally LO-criticality tasks in the LO-mode. For each step, a bin-packing strategy is used to assign tasks.

Bin-packing() in Algorithm 4.2 is used for assigning tasks in a certain mode M according to given fitting criteria *FitCriteria*, such as Worst-Fit or Best-Fit. For the partition in the HI-mode (i.e., $M = HI$), tasks are first

sorted by decreasing HI-mode utilization, and processors are sorted according to *FitCriteria*. Then, tasks are assigned one-by-one. Before a task τ_i is assigned to a processor, the schedulability of the processor (with τ_i assigned to it) needs to be checked. We only need to check the schedulability of the processor in the HI mode by using SCHEDULABLE-HI(); for the partition in the LO mode (i.e., $M = LO$), tasks are sorted by decreasing LO-mode utilization and assigned one-by-one after checking the schedulability of the processor using SCHEDULABLE-LO-and-MC().

The algorithm *DPM()* can use any bin-packing strategy. By changing the task sorting criteria, the fitting criteria for HI-criticality tasks (*Fitcriteria1* in Algorithm 4.1) and for LO-criticality tasks (*Fitcriteria2*), different algorithms can be obtained. Experimental results showed that using Worst-Fit for partitioning HI-criticality tasks produces the best results, as it distributes the HI-criticality tasks among all processors evenly, leaving more freedom to partition the LO-criticality tasks among a larger number of processors. For LO-criticality tasks, we observed that using First-Fit achieves maximum schedulability for these tasks.

Second phase - Optimization: Phase 2 is the optimization phase which takes the partitioning produced in phase 1 (if it succeeds) and reduces the number of migrations required in the mode change. This is done by changing the assignment of some LO-criticality tasks in the LO-mode. The tasks that migrate in the mode change are put in the list $\mathcal{MT}$ and sorted by decreasing LO-utilization. Then for each task τ_i in $\mathcal{MT}$, two attempts are made to relocate the task. First, we attempt to reassign the task in the LO-mode to the

Algorithm 4.1: DPM(Γ, P, *FitCriteria1*, *FitCriteria2*)

```
1: [Γ_HI, Γ_LO] = Split(Γ)
2: Phase 1.1: assign HI-criticality tasks
3: if Bin-packing(Γ_HI, P, FitCriteria1, HI)== FAIL then
4:     return FAIL
5: end if
6: Phase 1.2: assign LO-criticality tasks in HI-mode
7: if Bin-packing(Γ_LO, P, FitCriteria2, HI)== FAIL then
8:     return FAIL
9: end if
10: HI-partition=current partition
11: Phase 1.3: assign LO-criticality tasks in LO-mode
12: DeAllocateTasks(Γ_LO)
13: if Bin-packing(Γ_LO, P, FitCriteria2, LO)== FAIL then
14:     return FAIL
15: end if
16: Phase 2: optimize allocation
17: MT = tasks τ_i with π_i(LO) != π_i(HI) sorted by decreasing LO-utilization
18: for each τ_i ∈ MT do
19:     if SCHEDULABLE-LO-and-MC(τ_i,π_i(HI)) then
20:         assign(τ_i, π_i(HI))
21:         MT.update()
22:     end if
23: end for
24: for each τ_i, τ_j ∈ MT do
25:     if  π_i(HI) == π_j(LO) then
26:         if SCHEDULABLE-LO-and-MC(τ_i, π_i(HI) − τ_j)) and
            SCHEDULABLE-LO-and-MC(τ_j, π_i(LO) − τ_i)) then
27:             swap(τ_i,τ_j)
28:             MT.update()
29:         end if
30:     end if
31: end for
32: LO-partition=current partition
33: return SUCCESS
```

same processor ($\pi_i(HI)$) to which it is assigned in the HI-mode (Algorithm 4.1, lines 18–23). If this fails, another attempt is made to swap the task τ_i with another migrating task τ_j from $\pi_i(HI)$ such that $\pi_j(LO)$=$\pi_i(HI)$ but $\pi_j(HI)$ $\neq$ $\pi_j(LO)$ (lines 24–31).

We divide the schedulability analysis into two functions:

Algorithm 4.2: Bin-packing(Γ, P, *FitCriteria*, M)

```
1: Sort(Γ, Decreasing M-utilization)
2: Sort(P, FitCriteria)
3: for each τ_i ∈ Γ do
4:    for each p_j ∈ P do
5:       if M == HI then
6:          Sched = SCHEDULABLE-HI(τ_i,p_j)
7:       else
8:          Sched = SCHEDULABLE-LO-and-MC (τ_i,p_j)
9:       end if
10:      if Sched then
11:         assign(τ_i, p_j)
12:         Sort(P, FitCriteria)
13:         nextTask
14:      end if
15:   end for
16:   if τ_i not assigned on any processor then
17:      return FAIL
18:   end if
19: end for
20: return SUCCESS
```

SCHEDULABLE-HI(), used in the HI-partition, and SCHEDULABLE-LO-and-MC(), used in the LO-partition. If the dual-partitioned approach is not used, the conditions for schedulability for both the HI-mode and LO-mode need to be satisfied for the same partitioning.

4.2.1 Schedulability Analysis

Schedulability in LO-partition (SCHEDULABLE-LO-and-MC()): Since we also assume jobs from LO-criticality tasks are dropped during mode change, the AMC-rtb analysis in [46] can be reused for the schedulability test in LO-partition to check the schedulability in the LO-mode and the mode change.

Schedulability in HI-partition (SCHEDULABLE-HI()): The task set on the processor during the HI-partition is taken into account. All tasks are

checked using their HI-mode parameters. The Worst Case Response Time (WCRT) for task τ_i in the HI-partition is given by:

$$R_i(HI) = C_i(HI) + \sum_{j \in hp^{HI}(i)} \left\lceil \frac{R_i(HI)}{T_j(HI)} \right\rceil \cdot C_j(HI) \tag{4.1}$$

where $hp^{HI}(i)$ denotes the set of tasks which have higher priority than τ_i on the same processor in the HI-partition.

4.3 Experimental Evaluation

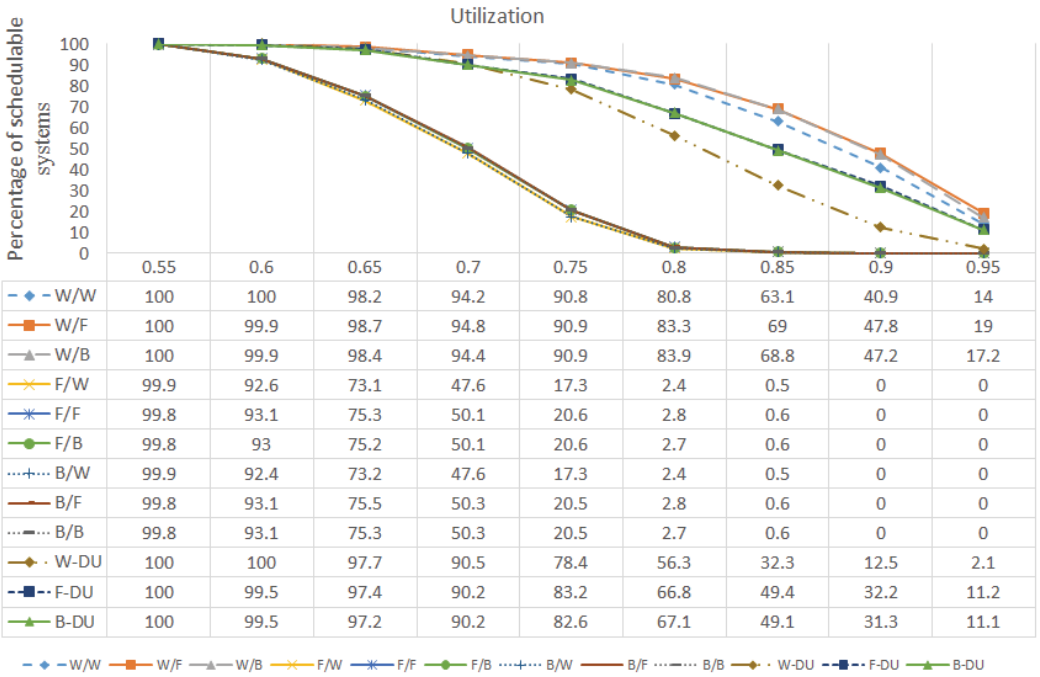
In this section, we compare different partitioning algorithms and evaluate the effectiveness of applying the proposed DPM approach to them. We will use the notation X/Y to denote the different algorithms where X is the bin-packing heuristic used for HI-criticality tasks and Y the heuristic for LO-criticality tasks, where X,$Y \in$ {W (Worst-Fit), F (First-Fit), B (Best-Fit)}. For example, W/B is the algorithm obtained by applying Worst-Fit to HI-criticality tasks, then Best-Fit to LO-criticality tasks.

Unless otherwise stated, for all experiments, the periods of the tasks (for LO-criticality tasks, this refers to T(LO)) are randomly selected from the set {10, 20, 40, 50, 100, 200, 400, 500, 1000}ms with uniform distribution. All LO-criticality tasks were guaranteed by default a maximum period in the HI-mode equal to double their nominal period ($T_i(HI) = 2 * T_i(LO)$). The $CFactor$ for HI-criticality tasks defining the ratio between HI-mode WCET and LO-mode WCET was randomly selected in [1-3] with uniform distribution. Then, the LO-utilization of tasks was generated using the UUnifast [101] algorithm with

the LO-utilization capped at 0.49 per task and the HI-utilization capped at 0.66 per task. The UUnifast algorithm generates tasks such that their total LO-mode utilization is equal to a given value (default=0.85*M). The HI-mode utilization depends on other random parameters ($Cfactor$ and $T(LO)$). Any system with a total HI-mode utilization $> M$ (number of processors) was discarded. The execution time of the task ($C_i(LO)$) is then derived from the period ($T(LO)$) and LO-utilization. $C_i(HI)$ is derived from $C_i(LO)$ and $Cfactor$. By default, half of the tasks in the system are HI-criticality tasks. 1000 systems were generated at each data point.

In the first experiment, we compare the algorithms before applying DPM, which includes nine different combinations of the partitioning algorithms (W/W, W/B, W/F, B/W, B/B, B/F, F/W, F/B, F/F). Furthermore, we also report the results of using the original bin-packing heuristic, which sorts tasks by decreasing utilization (DU), thus ignoring criticality. Figure 4.2 shows the percentage of schedulable systems as a function of the average per-processor LO-utilization. For this experiment, 40 tasks (20 LO, 20 HI) were scheduled on 4 processors.

It is clear from the figure that algorithms that assign HI-criticality tasks using Worst-Fit strategy produce substantially better results than the remaining ones. Among these, using First-Fit for LO-criticality tasks has the best schedulability. Interestingly, this coincides with previous results in [49] for EDF-VD [50] scheduled systems. In the remaining part of this section, we will focus on the W/F algorithm due to its superior performance and show the improvements that can be obtained by applying the DPM algorithm to



	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95
W/W	100	100	98.2	94.2	90.8	80.8	63.1	40.9	14
W/F	100	99.9	98.7	94.8	90.9	83.3	69	47.8	19
W/B	100	99.9	98.4	94.4	90.9	83.9	68.8	47.2	17.2
F/W	99.9	92.6	73.1	47.6	17.3	2.4	0.5	0	0
F/F	99.8	93.1	75.3	50.1	20.6	2.8	0.6	0	0
F/B	99.8	93	75.2	50.1	20.6	2.7	0.6	0	0
B/W	99.9	92.4	73.2	47.6	17.3	2.4	0.5	0	0
B/F	99.8	93.1	75.5	50.3	20.5	2.8	0.6	0	0
B/B	99.8	93.1	75.3	50.3	20.5	2.7	0.6	0	0
W-DU	100	100	97.7	90.5	78.4	56.3	32.3	12.5	2.1
F-DU	100	99.5	97.4	90.2	83.2	66.8	49.4	32.2	11.2
B-DU	100	99.5	97.2	90.2	82.6	67.1	49.1	31.3	11.1

Fig. 4.2: Percentage of schedulable systems at different LO-utilizations for various heuristics (before applying DPM)

it (W/F-DPM). Using DPM on other algorithms generates similar improvements.

Figure 4.3 shows the improvement obtained by applying the proposed approach on the W/F algorithm. The W/F-DPM algorithm (Algorithm 4.1 with *FitCriteria1*=Worst-Fit and *FitCriteria2*=First-Fit) schedules more systems than the conventional W/F algorithm, especially at high utilizations. At utilizations of 0.8 and higher, the W/F-DPM enhances the schedulability of W/F by 17%, increasing at utilizations of 0.9 and beyond to 28%.

To study the robustness of the DPM algorithm at various parameter settings, the key experiment parameters were varied in the subsequent experiments. The average per-task utilization was varied by varying the total task

count in the range [20-100] while keeping the LO-utilization fixed at 0.85 per processor. All other parameters are kept the same. Figure 4.4 shows the percentage of schedulable systems as a function of the total task count (higher task counts indicate lower average task utilizations). W/F-DPM increases the schedulability of W/F at all task utilizations by an average of 20%. This percentage is higher for systems with a higher number of tasks.

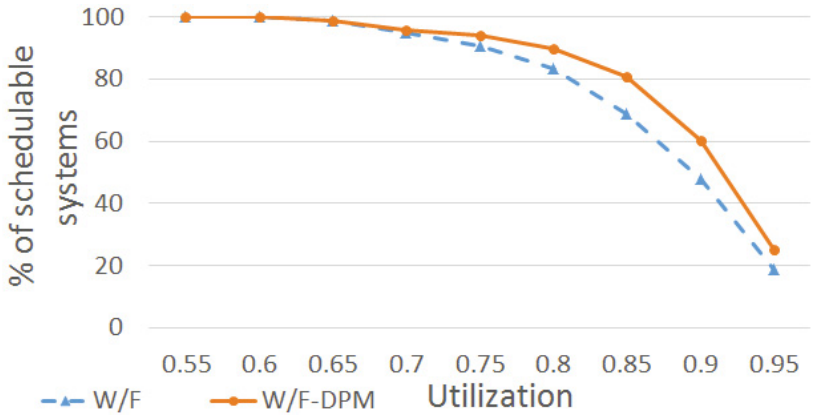


Fig. 4.3: Improvement on schedulability at different LO-utilizations

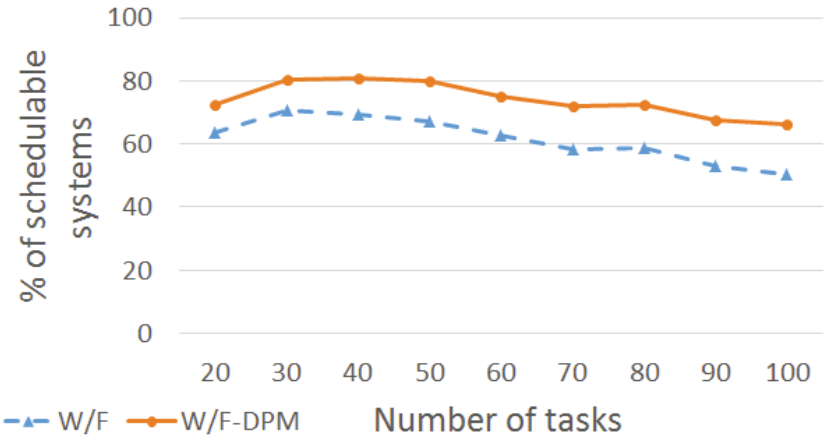


Fig. 4.4: Improvement on schedulability at different tasks counts

The impact of having a different HI/LO criticality mix is studied in Figure 4.5. The percentage of HI-criticality tasks is varied in the range [20%-70%]

while keeping all other parameters fixed. For systems with ≤30% HI-criticality tasks, almost all systems are schedulable. With more HI-criticality tasks in the mix, the performance of both algorithms degrade but W/F-DPM performs better. For these systems, W/F-DPM schedules 67% of systems while W/F schedules 56%. As shown in Figure 4.6, similar observation can be drawn for varying numbers of processors.

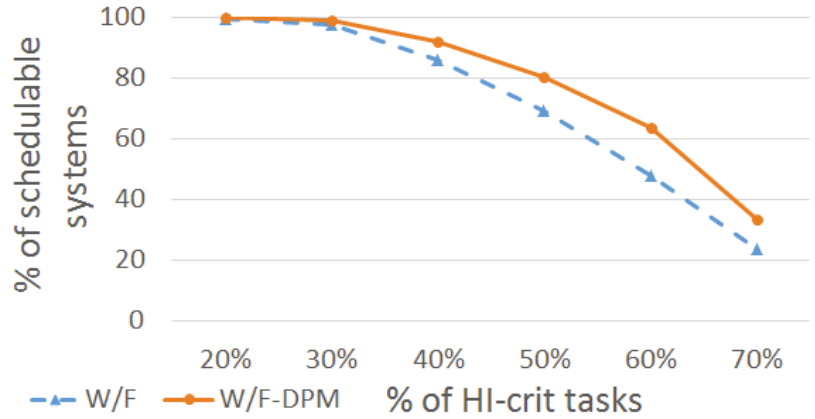


Fig. 4.5: Improvement on schedulability at different percentages of HI tasks

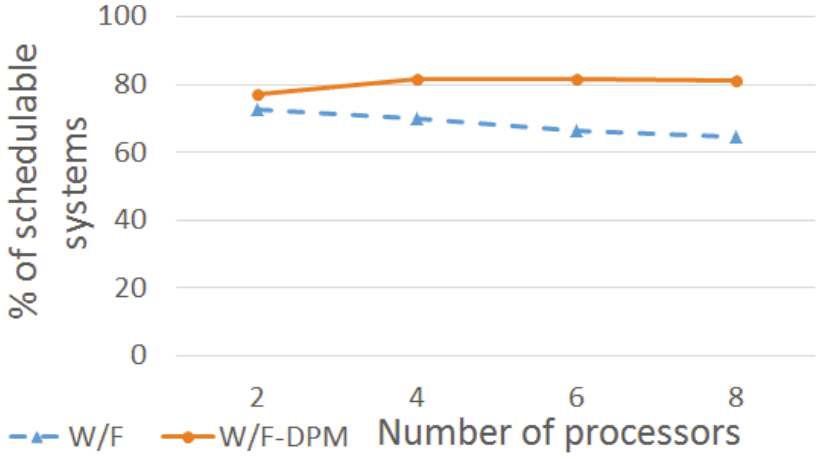


Fig. 4.6: Scalability of DPM to larger numbers of processors

In all previous experiments, the ratio of HI-mode WCET to LO-mode

WCET ($CFactor$) was randomly selected in [1-3]. To check whether changing the range of HI-criticality WCET impacts schedulability, the maximum value of $CFactor$ (3 in previous experiments) was varied. Figure 4.7 shows that schedulability drops, as expected, when the maximum allowed C(HI)/C(LO) ratio increases. However, the schedulability of DPM drops more moderately. In the last experiment, the effect of providing different levels of minimum service guarantees to different LO-criticality tasks in the HI-mode is explored. Instead of having a fixed value for the ratio $T(HI)/T(LO)$ as in the previous experiments, this ratio was chosen randomly in the range [1-T_{max}] with uniform distribution, and T_{max} was varied in the range [2-7]. DPM maintained on average a 16% schedulability advantage in line with the results of the previous experiments.

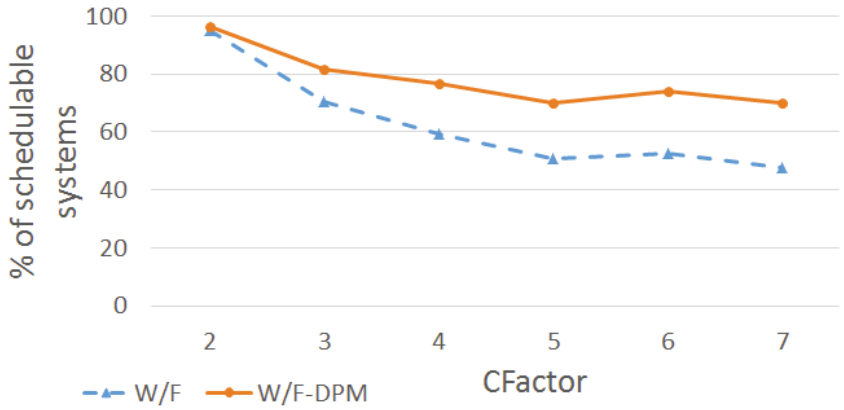


Fig. 4.7: Improvement at different values of the maximum CFactor

To evaluate the impact of Phase 2 of the algorithm in reducing the number of migrations, we performed an experiment to check the number of migrations before and after this phase at different utilizations. We used the same parameters as in the experiment in Figure 4.3 and averaged the results over 1000

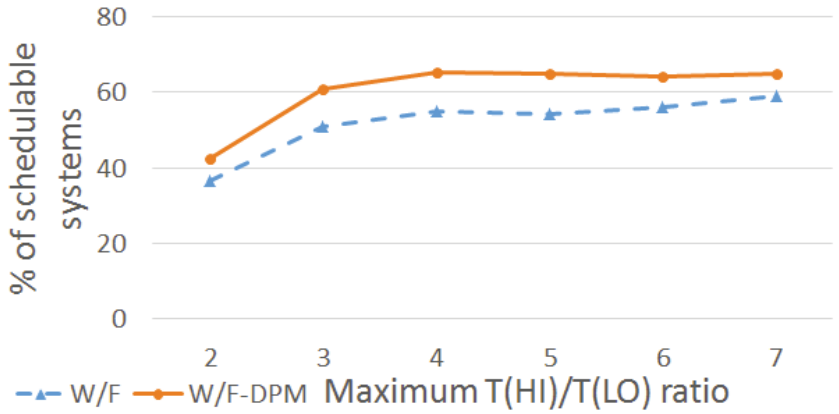


Fig. 4.8: Improvement at different values of the maximum T(HI)/T(LO) ratio

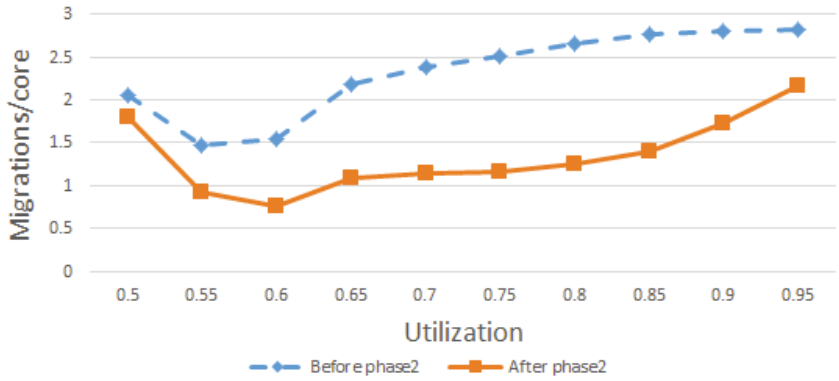


Fig. 4.9: Impact of Phase 2

schedulable systems at each utilization. The results are shown in Figure 4.9. The results show that Phase 2 reduces the number of migrations by 42% on average. After the application of phase 2, an average of only 1.4 migrations are required on each core which reduces the mode change overhead.

4.4 Conclusion

In this work, we present a study of fixed-priority partitioned scheduling on multicore architectures. We compare the performance of bin-packing heuristics when applied to elastic mixed-criticality systems. We also present a novel

dual-partitioned mixed-criticality scheduling algorithm for multicore architectures. The dual partitioned approach is applicable to any bin-packing heuristic and can increase the schedulability of the heuristic on a given platform while providing minimum service guarantees to LO-criticality tasks. Experimental results show that at utilizations of 0.8 or higher, schedulability is enhanced by 17%. The proposed approach thus optimizes the use of the available resources by scheduling more functionality on the same hardware, and can potentially lead to significant savings in the computing resources.

Chapter 5

Tolerating Hardware Faults in Mixed-Criticality Systems

One of the main objectives of the current research on MCS is to develop systems that are efficient while at the same time being certifiable and adhere to safety standards such as ISO26262. While the criticality and Worst Case Execution Time (WCET) related part of the MCS design problem received much attention, the reliability of these systems which do contain a safety critical part was not similarly addressed. These same standards specify reliability requirements for systems. An MCS system must satisfy both sets of requirements: criticality and reliability. This Chapter[1] provides a comprehensive study of fault-tolerance in Mixed-Criticality Systems (MCS) for both tran-

[1]The work on transient faults in this chapter was done in close collaboration with Jonah Caplan. The core model was co-developed with Mr. Caplan. I developed the schedulability analysis and implemented the simulation framework for single cores and lockstep architectures. Mr. Caplan extended the work to other fault-tolerant mechanisms to which I contributed the derivation of the generic re-execution profiles. The work on permanent faults did not involve external collaboration.

sient and permanent faults. We focus on hardware faults in this chapter. To enable the design of fault-tolerant MCS, we make the following contributions

1. We propose a novel four-mode MCS model and accompanying schedulability analysis to simultaneously consider certification, transient faults, and the Quality of Service (QoS) provided to LO-criticality tasks (Sections 5.3, 5.4). Our model (a) differentiates between requirements imposed by certification and fault tolerance, (b) provides suitable guarantees for HI tasks in each case, and (c) optimizes the QoS provided to LO tasks.

2. We make the observation that On-Demand Redundancy (ODR), which allows cores to be dynamically coupled and decoupled at runtime, can potentially have significant advantages for MCS. Based on this, we generalize the four-mode model and analysis to support ODR with the various fault-tolerance mechanisms (discussed in Section 2.5.1). A task set transformation is proposed to generate a modified task set that can support different mechanisms while satisfying reliability and certification requirements (Section 5.5).

3. We propose a Design Space Exploration (DSE) approach that uses the generalized analysis and supports ODR and heterogeneous platforms (Section 5.6). The design variables explored are task mappings and selection of redundancy techniques. The objective is to produce reliable and schedulable systems while maximizing the LO tasks QoS. Experiments show that ODR can improve QoS provided to non-critical tasks by 29%

on average, compared to lockstep execution.

4. For tolerating permanent faults, we present a design approach for reliable MCS scheduled with partitioned fixed-priority scheduling under permanent processor failures. To this end, we extend the standard mixed-criticality model and schedulability analysis to support the failure of one processor in the system (Sections 5.9, 5.10).

5. We propose a Mixed Integer Linear Programming (MILP) based DSE approach for MCS that can tolerate permanent faults (Section 5.11). Reliability is pro-actively considered as part of the initial design problem. The MILP formulation finds, if it exists, a feasible: (a) task allocation, (b) priority assignment, and (c) alternative assignment for each HI-criticality task if the core to which it is initially assigned fails. Experiments illustrate the advantage of this pro-active approach to reliability compared to a baseline 2-step exploration process. For example, in systems composed of 20 tasks and 4 cores, a 3.2X improvement in schedulability is observed.

5.1 Scheduling MCS with Transient Faults: Motivation

As discussed in Section 2.5, researchers have begun to evaluate MCS in the context of transient faults. However, they often treat in the same way task overruns (resulting from optimistic WCET estimates) and re-executions (resulting from fault mitigation). For correct and efficient implementation, the two scenarios must be differentiated, and sufficient—but not excessive—

guarantees must be provided in each case. The work in [68] differentiates each source of delay, but conservatively drops all LO tasks when either event occurs. Dropping LO criticality tasks in MCS has drawn considerable concerns from system engineers [45]. Systems integrators are increasingly having to implement more features with fewer resources. MCS must therefore provide as much QoS to LO tasks as possible.

In this work, we propose a novel, four-mode model for MCS that addresses both certification and reliability requirements, while retaining as many LO tasks as possible when either overruns or transient faults occur. The proposed model differentiates between transient faults (TF mode), execution time overruns (OV mode), and their combination (HI mode). Transient faults require re-execution, and overruns require task-dependent increases in execution budget. When fault-tolerant MCS are implemented in two modes as in [68], designers must assume these events, though different in their rates of occurrence and consequences for task execution, always coincide. This results in overly pessimistic restrictions on HI mode. Our experimental results demonstrate that for single-core platforms, the new modes (TF, OV) of the four-mode system improve LO task QoS by 20.2% and 42.9% respectively compared to a two-mode model. Extended to partitioned, multi-core systems, the benefits of the four-mode model are even more substantial.

5.2 Assumptions and Notation

5.2.1 Task model

A system in our model consists of a set of N sporadic mixed-criticality tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$, and a supporting architecture which can be single-core or multi-core consisting of $L \geq 1$ processors. We assume that each core p is characterized by a failure rate λ_p (failures per unit time). The task set Γ can be divided into two subsets: high criticality tasks (Γ_{HI}), and low criticality tasks (Γ_{LO}). We further assume that (a) tasks are independent (i.e., no blocking due to shared resources), (b) tasks do not suspend themselves, other than at the end of their execution, (c) the overheads due to context switching, migration, etc. are negligible. Each task τ_i has a 6-tuple of parameters $\langle L_i, C_i(LO), C_i(HI), T_i, D_i, \pi_i \rangle$, where:

- $L_i \in$ {LO, HI} denotes the criticality level of τ_i.
- $C_i(LO)$ is the designer-specified WCET for τ_i.
- $C_i(HI)$ is the WCET specified by certification authority.
- If $L_i = LO$, $C_i(HI) = C_i(LO)$; otherwise $C_i(LO) \leq C_i(HI)$.
- T_i denotes the minimum inter-arrival time of τ_i.
- Tasks have implicit deadlines, i.e., $D_i = T_i$.
- π_i denotes the priority of τ_i.

$u_i = C_i/T_i$ is the utilization task of τ_i. Since tasks can have two C values, there will be two corresponding utilization values: $u_i(LO)$, and $u_i(HI)$.

Tasks are scheduled with fixed priority, and, if the architecture is multicore, with partitioned scheduling. Each core implements a local scheduler. A global priority order is enforced among all tasks regardless whether allocated to the same core or not. Lower values indicate higher priorities.

5.2.2 Failure probabilities for transient faults

Each criticality level is characterized by a maximum probability of failure, given as a probability of failure per hour (PFH). Temporal and/or spatial redundancy are used as the hardening mechanisms to achieve the required reliability level. Each HI task τ_i must be guaranteed to be schedulable even when it re-executes in response to faults to achieve a failure rate of at most $PFH_i = PFH(HI)$. If a task has replicas on more than one core, all replicas must be able to execute and re-execute if necessary before the task's deadline. For LO-criticality tasks, we assume that they do not have specific PFH requirements, hence they do not need to have temporal or spatial redundancy.

5.3 The Four-Mode System Model

To handle both task overruns and faults in the system efficiently, we propose a system model with four system modes for tasks with two criticality levels. In MCS, we cannot know *a priori* whether the task will actually overrun its $C(LO)$. These events are generally rare. Similarly, when considering faults, we cannot know in advance if the task will experience a transient error during its execution. If a task experiences a fault, it must be re-executed. These re-executions are also rare, and have a different effect on execution time.

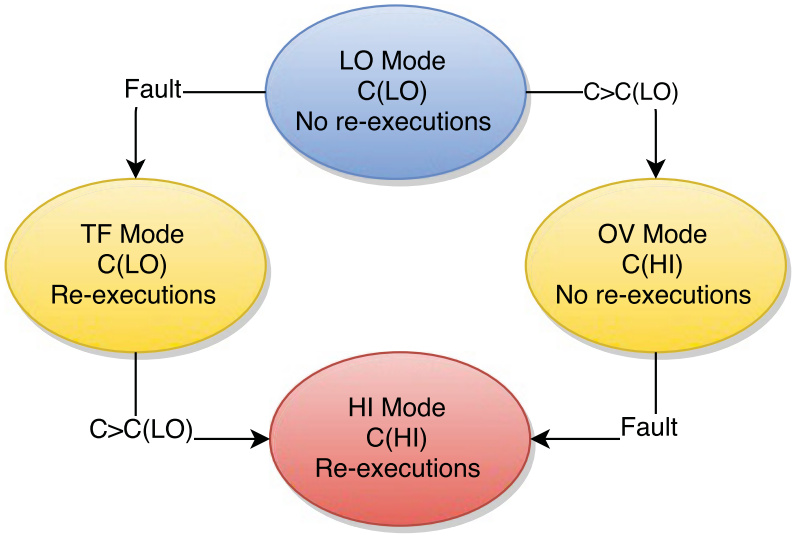


Fig. 5.1: The four-mode system model

Figure 5.1 illustrates the four system modes and possible mode changes. The modes are denoted LO (low), TF (transient faults), OV (task overruns), and HI (high). We denote the system mode by S, $S \in \{LO, TF, OV, HI\}$. The rationale for having four system modes, as opposed to the regular two-mode model of MCS in [46, 68], is to distinguish between these two different events (task overruns and transient faults) and provide suitable guarantees and QoS in each case. From the original LO mode, each of these two independent events will cause the system to enter a different mode with sufficient guarantees (sufficient time for re-execution or overruns) for the corresponding type of event. To be safe, a fourth mode, HI, is used if an overrun occurs while the system is addressing a fault in TF, or vice versa. Most existing work on MCS perform forward mode changes only (from a lower mode to a higher mode) [12]. We adopt a similar strategy here, and defer reverse mode changes to future work.

5.3.1 Re-execution requirements in lockstep cores

For lockstep, a transient fault results in a re-execution on the same core. We denote the number of executions (including the initial one) required for a HI task τ_i to achieve a certain reliability (PFH) level in mode S as $n_i(S)$. The probability of failure per job of τ_i and the failure rate for a core per τ_i's execution can be obtained by scaling PFH_i and λ_{p_i} respectively where p_i denotes the core to which the task τ_i is assigned. Considering each execution of task τ_i as independent, then Constraint (5.1) must be satisfied for TF and HI modes. As a result, $n_i(S)$ is given by Equation (5.2).

$$PFH_i \cdot T_i \geq (\lambda_{p_i} \cdot C_i(S))^{n_i(S)} \tag{5.1}$$

$$n_i(S) = \begin{cases} \left\lceil \dfrac{\log(PFH_i \cdot T_i)}{\log(\lambda_{p_i} \cdot C_i(S))} \right\rceil, & S \in \{TF, HI\} \\ 1, & S \in \{LO, OV\} \end{cases} \tag{5.2}$$

The variable n_i can be written as a vector for all four modes (LO, TF, OV, HI) which we refer to as the *re-execution vector* N_i. For lockstep, we have:

$$N_i^{LS} = \langle 1, n_i(TF), 1, n_i(HI) \rangle \,. \tag{5.3}$$

Other mechanisms discussed in Section 2.5.1 combine both temporal and spatial locality. Their re-execution profiles can differ from lockstep. We derive their re-execution vectors in Section 5.5.

5.3.2 System operation

Our system operates as follows:

1. The system starts in the LO mode, where all tasks i execute once ($n_i(LO) = 1$) and can run up to $C_i(LO)$.

2. In LO mode, if any task i executes beyond $C_i(LO)$, the system moves into overrun (OV) mode, where all HI-criticality tasks j can safely execute once ($n_j(OV) = 1$) up to $C_j(OV) = C_j(HI)$.

3. Alternatively, in LO mode, if any HI-criticality task experiences a transient fault, the system moves into transient fault (TF) mode, where each HI-criticality task i is guaranteed a sufficient number of re-executions to satisfy their reliability requirements ($n_i(TF)$). *During the mode transition, the required re-executions for a HI task must be guaranteed to finish before its deadline.* In this way, reliability requirements are met even when the task starts its execution in a mode that does not consider re-executions. The original task and all re-executions are allowed to execute up to $C_i(TF) = C_i(LO)$.

4. If any of the (re-)executions of task i in TF mode overruns $C_i(LO)$, the system moves into HI mode. A move to this mode is also possible if the system is in OV mode and a transient fault occurs. In HI mode, HI-criticality tasks i can execute up to $C_i(HI)$ and re-execute $n_i(HI) \geq n_i(TF)$ times.

Modes OV and TF cover the basic cases of task overruns and transient faults respectively, by dropping some LO-criticality tasks to allow more time

for HI tasks. Both events leading to HI have a low probability, however for highly critical systems, it is important to consider all cases. It is worth noting that in this mode, more re-executions for a HI-criticality task i might be needed than in TF mode ($n_i(HI) \geq n_i(TF)$), because HI-criticality tasks are assumed to run longer ($C_i(HI) \geq C_i(TF)$) and hence could experience more errors.

5.3.3 Providing QoS to LO-criticality tasks

One of the concerns on the two-mode MCS model (LO and HI) [46, 68] is the assumption that all LO tasks are dropped in HI mode [12]. Our proposed four-mode model does not drop all LO tasks in the modes OV, TF, and HI. Instead, LO tasks are selectively allowed to run in the new mode as long as this does not affect the schedulability of HI tasks. New schedulability analysis that extends to the new model is presented in Section 5.4. This analysis provides offline guarantees to all HI tasks and the selected subset of LO tasks. In this way, the designer can ensure the schedulability of the task set regardless of the runtime scenario. In this work, we simply maximize the number of LO tasks scheduled in each mode. Alternatively, we may take into consideration the designer's preferences.

5.3.4 An illustrative example

To illustrate the benefit of the proposed model, we apply the proposed four-mode model to the example task set in Table 5.1. The tasks are arranged in priority order (the lower index, the higher priority). We assume the number of executions for HI task τ_1 is $n_1(TF) = n_1(HI) = 3$. If an overrun occurs in

Table 5.1: An Example Task Set

Task	$C(LO)$	$C(HI)$	T=D	L
τ_1	3	4	12	HI
τ_2	4	-	12	LO
τ_3	4	-	12	LO
τ_4	1	-	12	LO

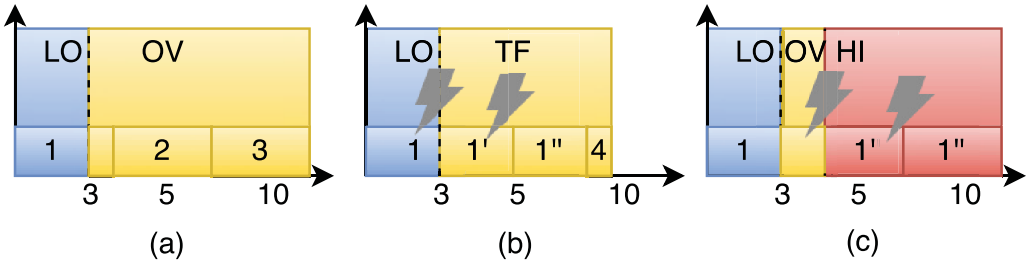


Fig. 5.2: An execution trace for the task set in Table 5.1 when (a) an overrun occurs; (b) a fault occurs; (c) both occur.

task τ_1 (Fig. 6.2a), the system moves into OV mode, where $12 - 4 = 8$ time units are available to run the LO-criticality tasks in the system. Therefore, we only need to drop one task (τ_4, for example). If τ_1 needs to re-execute due to faults (Fig. 6.2b), the system can still schedule one LO task (τ_4), since $n_1(TF) = 3$. If both fault and overrun occur (Fig. 6.2c), the system switches to HI mode, and τ_1 can still run safely with all re-executions. For a two-mode model, any overrun or fault will lead to a scenario similar to Fig. 6.2c, and all LO tasks will be dropped immediately. The four-mode model clearly improves LO task QoS over the two-mode model.

5.4 Schedulability Analysis with Transient Faults

Schedulability analysis for the four-mode system is derived by extending the Adaptive Mixed Criticality response time bound (AMC-rtb) analysis, proposed for the two-mode model with AMC scheduling [46]. We first present the basic analysis assuming that lockstep cores are the deployment platform and that sequential re-execution is the only fault tolerance mechanism. In Section 5.5, we extend it to ODR with a range of platforms.

Under the basic analysis, a system is schedulable if:

1. HI tasks must be schedulable in all four modes;
2. LO tasks must be schedulable in the LO mode.

Therefore, for modes OV, TF, and HI we only need to check the schedulability of HI tasks. It is not required to guarantee the schedulability of any LO task in these three modes. However, through DSE, we try to schedule LO tasks in these modes as long as the system schedulability is not affected.

For LO mode the response time of a task τ_i is given by:

$$R_i^{(LO)} = C_i(LO) + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(LO)}}{T_j} \right\rceil \cdot C_j(LO) \tag{5.4}$$

where $hp(i)$ is the set of tasks on τ_i's core that have higher priority than τ_i (including both HI and LO tasks). The DSE algorithm dynamically re-allocates tasks. Therefore, the set $hp(i)$ is dynamically changing during DSE, and it shall reflect the current task set on τ_i's core.

When evaluating schedulability in mode TF, we need to consider two cases. First, the stable mode where all tasks released before the mode change have either finished execution or been dropped. Second, we need to verify that the mode change from LO to TF does not cause the given task to miss its deadline. The worst case response time of a task in the mode change is larger than that in the stable mode: In addition to the task τ_i under analysis and higher priority tasks executing in the stable mode, we also need to consider the effect of LO-criticality tasks that may be dropped in TF but have already delayed τ_i's execution. The worst case response time of a task in TF is hence the response time of the mode change, and is given by Equation (5.5).

$$\begin{aligned} R_i^{(TF)} &= n_i(TF) \cdot C_i(LO) \\ &+ \sum_{j \in hpC(TF,i)} \left\lceil \frac{R_i^{(TF)}}{T_j} \right\rceil \cdot n_j(TF) \cdot C_j(LO) \\ &+ \sum_{k \in hp(i) \setminus hpC(TF,i)} \left\lceil \frac{R_i^{(LO)}}{T_k} \right\rceil \cdot C_k(LO) \end{aligned} \tag{5.5}$$

In TF, we must account for task re-executions for HI-criticality tasks j, hence $C_j(LO)$ is multiplied by the number of re-executions $n_j(TF)$. $hpC(TF, i)$ is the set of continuing tasks (i.e., tasks that continue to execute in the current mode TF) that have higher priority than τ_i. This set of tasks contains all HI tasks and those LO tasks selected to continue. The last summation considers the tasks that are dropped in the LO-to-TF mode change (in the set $hp(i) \setminus hpC(TF, i)$). These tasks can only execute up to $R_i^{(LO)}$ since a mode change must occur before $R_i^{(LO)}$. Otherwise τ_i would have terminated before the mode change. Since only LO tasks can be dropped, we use $C_k(LO)$ and

$n_k = 1$ here.

The response time for mode OV can be similarly calculated. However, HI tasks can execute up to $C(HI)$ and no re-executions are considered ($n = 1$ for all tasks). The response time in mode OV is given by:

$$R_i^{(OV)} = C_i(L_i) + \sum_{j \in hpC(OV,i)} \left\lceil \frac{R_i^{(OV)}}{T_j} \right\rceil \cdot C_j(L_j) + \sum_{k \in hp(i) \setminus hpC(OV,i)} \left\lceil \frac{R_i^{(LO)}}{T_k} \right\rceil \cdot C_k(LO) \tag{5.6}$$

In mode HI, we need to consider both overruns ($C_i(HI)$) and re-execution ($n_i(HI) \geq 1$). Transitions to the HI mode can originate from either TF or OV mode. Both mode changes need to be verified. The worst case scenario occurs when we encounter two consecutive mode changes in quick succession (LO-to-TF-to-HI or LO-to-OV-to-HI).

Focusing on the LO-to-TF-to-HI mode change first (Equation (5.7)), the task τ_i under analysis and the high priority continuing tasks in mode HI $hpC(HI, i)$ are allowed to execute $n(HI)$ times (first and second terms on the right hand side of the equation). For tasks that are allowed to continue in TF mode but dropped in HI mode (i.e., in the set $hpC(TF, i) \setminus hpC(HI, i)$), we assume that the mode change from TF to HI has to happen before $R_i^{(TF)}$ (third term). Finally, for the remaining tasks in $hp(i)$ (i.e., tasks that started in LO and got dropped in TF), the interference is bounded by the fact that the mode change from LO to TF has to happen before $R_i^{(LO)}$ (last term). Note that we do not consider re-executions for those dropped tasks ($n_k = n_l = 1$)

as they are all LO-critical.

$$\begin{aligned} R_i^{(HIa)} &= n_i(HI) \cdot C_i(L_i) \\ &+ \sum_{j \in hpC(HI,i)} \left\lceil \frac{R_i^{(HIa)}}{T_j} \right\rceil \cdot n_j(HI) \cdot C_j(L_j) \\ &+ \sum_{k \in hpC(TF,i) \setminus hpC(HI,i)} \left\lceil \frac{R_i^{(TF)}}{T_k} \right\rceil \cdot C_k(LO) \\ &+ \sum_{l \in hp(i) \setminus hpC(TF,i)} \left\lceil \frac{R_i^{(LO)}}{T_l} \right\rceil \cdot C_l(LO) \end{aligned} \tag{5.7}$$

Similarly, the task response time of τ_i for LO-to-OV-to-HI transition is presented in Equation (5.8):

$$\begin{aligned} R_i^{(HIb)} &= n_i(HI) \cdot C_i(L_i) \\ &+ \sum_{j \in hpC(HI,i)} \left\lceil \frac{R_i^{(HIb)}}{T_j} \right\rceil \cdot n_j(HI) \cdot C_j(L_j) \\ &+ \sum_{k \in hpC(OV,i) \setminus hpC(HI,i)} \left\lceil \frac{R_i^{(OV)}}{T_k} \right\rceil \cdot C_k(LO) \\ &+ \sum_{l \in hp(i) \setminus hpC(OV,i)} \left\lceil \frac{R_i^{(LO)}}{T_l} \right\rceil \cdot C_l(LO) \end{aligned} \tag{5.8}$$

The response time of τ_i in HI mode is the maximum of the two possible transitions:

$$R_i^{(HI)} = \max(R_i^{(HIa)}, R_i^{(HIb)}) \tag{5.9}$$

Table 5.2: An Example Task Set

	$C(LO)$	$C(HI)$	T=D	n	L
τ_1	3	4	20	2	HI
τ_2	4	6	20	2	HI
τ_3	4	-	20	1	LO
τ_4	1	-	20	1	LO

5.4.1 Reducing model pessimism

It is unlikely that all tasks would be required to re-execute the maximum number of times within a given period. Transient faults are rare, and the number of faults can be limited with a parameter provided by the designer [68, 102]. Relaxing the assumption that failures are independent, let F be the maximum number of faults expected in any interval of length $D_{\max}$ where $D_{\max}$ is the largest relative deadline among the tasks in the task set. This parameter can be obtained through fault analysis considering the expected environmental conditions of the system.

If we limit the maximum number of additional re-executions that need to be considered in the response time calculation F, it is possible to further improve LO-criticality task QoS. For example, consider the task set in Table 5.2. Assuming tasks are sorted by priority order (task τ_1 has the highest priority). The response time of task τ_3 in mode TF assuming no tasks are dropped is $2 \times 3 + 2 \times 4 + 4 = 18$. If the designer knows that a maximum of one error can occur in 20 time units ($F = 1$), then only one job of either task τ_1 or τ_2 can fail within a single job of τ_3. The worst case occurs when τ_2 fails and the response time can be bounded at 14.

More generally, for modes TF and HI, the analysis can be formulated as an

ILP (Integer Linear Programming) problem, by slightly modifying and adding constraints to Eqs. (5.5), (5.7), and (5.8). We replace n_i (the number of re-executions required by task τ_i to meet its PFH requirement) by $1 + f_i$ where f_i is the number of re-executions we need to consider from τ_i in the worst case response time calculation. In the case of TF, we then maximize:

$$\begin{aligned} R_i^{(TF)} &= (1 + f_i) \cdot C_i(LO) \\ &+ \sum_{j \in hpC(TF,i)} \left\lceil \frac{R_i^{(TF)}}{T_j} \right\rceil \cdot (1 + f_j) \cdot C_j(LO) \\ &+ \sum_{k \in hp(i) - hpC(TF,i)} \left\lceil \frac{R_i^{(LO)}}{T_k} \right\rceil \cdot C_k(LO) \end{aligned} \tag{5.10}$$

under the constraints:

$$1 + f_i \leq n_i, \quad fi \geq 0 \, \forall \tau_i \tag{5.11a}$$

$$\sum_i f_i \leq F. \tag{5.11b}$$

Similar formulations can be derived for the transitions to HI, Eqs. (5.7), and (5.8), respectively. In practice, the maximum can be easily determined with polynomial complexity by sorting the list of tasks with $n > 1$ by descending utilization, then adding the maximum number of re-executions for each task in the list until there are F errors.

5.5 Generalization to ODR

We generalize the model and analysis presented in the previous sections to the range of mechanisms discussed in Section 2.5.1. When a mechanism

is applied to a given task set Γ, a transformation is needed to generate a modified task set Γ'. Γ' depends on the characteristics of the mechanism applied. *Each mechanism is characterized by three parameters*: number of replicas, re-execution profile N for each replica, and mapping constraints. The number of replicas (including the original task) for Dual Modular Redundancy (DMR), Triple Modular Redundancy (TMR), and Passive Replication (PR) is 2, 3, and 3, respectively. The re-execution profiles and mapping constraints for each mechanism are shown in Table 5.3. The second column shows the re-execution profiles for the different mechanisms. The tuple represents the number of executions required for a given mechanism to meet the system's reliability requirements in the four modes (LO, TF, OV, HI). $n_i(TF)$ and $n_i(HI)$ are derived according to Equation (5.2). The last column in Table 5.3 shows mapping constraints that must be added in order to properly reflect the semantics of the techniques. These constraints ensure that replicas of the same task are assigned to different cores.

We now explain how we derive the re-execution profiles such that each mechanism can recover from the same number of errors as lockstep execution (Equations (5.2) and (5.3)). In lockstep with sequential re-execution, x executions are needed to recover from x-1 errors. The same is true for DMR. Therefore, each DMR replica needs to be executed for the same number of times as in lockstep. For TMR, one fault does not require re-execution as the voter will have two correct values and can determine the correct output. In the worst case, two faults would require one re-execution and hence the TF and HI modes require $\lceil n_i(TF)/2 \rceil$ and $\lceil n_i(HI)/2 \rceil$ executions respectively. For PR,

Table 5.3: Transformation parameters for the four fault-tolerance mechanisms.

	Replicas & Their Profile (N)	Constraints
LS	$\tau_i \langle 1, n_i(TF), 1, n_i(HI)\rangle$	-
DMR	$\tau_i \langle 1, n_i(TF), 1, n_i(HI)\rangle$ $\tau_{i.1} \langle 1, n_i(TF), 1, n_i(HI)\rangle$	$p_i \neq p_{i.1}$
TMR	$\tau_i \langle 1, \lceil n_i(TF)/2\rceil, 1, \lceil n_i(HI)/2\rceil\rangle$ $\tau_{i.1} \langle 1, \lceil n_i(TF)/2\rceil, 1, \lceil n_i(HI)/2\rceil\rangle$ $\tau_{i.2} \langle 1, \lceil n_i(TF)/2\rceil, 1, \lceil n_i(HI)/2\rceil\rangle$	$p_i \neq p_{i.1} \neq p_{i.2}$
PR	$\tau_i \langle 1, 1, 1, 1\rangle$ $\tau_{i.1} \langle 1, 1, 1, 1\rangle$ $\tau_{i.2} \langle 0, n_i(TF) - 1, 0, n_i(HI) - 1\rangle$	$p_i \neq p_{i.1}$

two replicas execute one time in all modes. The other replica only executes if an error is detected. Therefore, it has 0 executions in LO and OV modes. In the unlikely event of another fault, it re-executes. Therefore we have $n_i(TF)-1$ and $n_i(HI) - 1$ executions in the TF and HI modes respectively.

Modified task sets are generated through a transformation from the original task set using the profiles in Table 5.3. All replicas inherit the parameters of their original tasks. An example of such a transformation is shown in Table 5.4. When applying DMR (Table 5.4b), one replica for the HI criticality task τ_1 is added and its parameters are copied from the original task. The re-execution profiles N for τ_1 and its replica remain the same as in the lockstep case. A mapping constraint is added to ensure that τ_1 and its replica are assigned to different cores. The LO task τ_2 is not affected by the transformation. TMR and PR transformations follow the same steps.

With this transformation, the operation of the system follows the steps discussed in Section 5.3.2 after replacing the n values with the modified values

Table 5.4: Task set transformation

(a) Example task set

	C(LO)	C(HI)	T=D	L	N
τ_1	5	10	25	HI	$\langle 1, 2, 1, 2 \rangle$
τ_2	5	-	20	LO	-

(b) DMR transformation (Constraint: $p_1 \neq p_{1.1}$)

	C(LO)	C(HI)	T=D	L	N
τ_1	5	10	25	HI	$\langle 1, 2, 1, 2 \rangle$
$\tau_{1.1}$	5	10	25	HI	$\langle 1, 2, 1, 2 \rangle$
τ_2	5	-	20	LO	-

(c) TMR transformation (Constraint: $p_1 \neq p_{1.1} \neq p_{1.2}$)

	C(LO)	C(HI)	T=D	L	N
τ_1	5	10	25	HI	$\langle 1, 1, 1, 1 \rangle$
$\tau_{1.1}$	5	10	25	HI	$\langle 1, 1, 1, 1 \rangle$
$\tau_{1.2}$	5	10	25	HI	$\langle 1, 1, 1, 1 \rangle$
τ_2	5	-	20	LO	-

(d) PR replication (Constraint: $p_1 \neq p_{1.1}$)

	C(LO)	C(HI)	T=D	L	N
τ_1	5	10	25	HI	$\langle 1, 1, 1, 1 \rangle$
$\tau_{1.1}$	5	10	25	HI	$\langle 1, 1, 1, 1 \rangle$
$\tau_{1.2}$	5	10	25	HI	$\langle 0, 1, 0, 1 \rangle$
τ_2	5	-	20	LO	-

for the appropriate mechanism as in Table 5.3. The schedulability of the new task set can be verified using the analysis presented in Section 5.4, which shall use the new n values. We note that all techniques have $n(LO)$ and $n(OV)$ values of either 0 or 1. When $n = 0$, the task replica is not executing and its schedulability is not considered.

5.6 DSE with Transient Faults

Design space exploration is done using a genetic algorithm (GA) that takes as input the modified task set and mapping constraints described in Section 5.5. The GA explores different task mappings and redundancy techniques. The objective is to find a hardened system configuration that satisfies certification and reliability requirements while maximizing QoS provided to LO tasks. The GA is divided into two nested stages as in [72]. It makes use of the new schedulability analysis to express arbitrary fault tolerance strategies on a per-task basis. The outer genetic algorithm explores techniques used to harden each task while the nested genetic algorithm explores the core assignment for a given fault tolerance configuration. The two stages of genetic algorithms (GA) are implemented using JGAP [103].

The basic flow is shown in Fig. 5.3. First, the Reliability Aware (RA) stage finds a fault tolerance mechanism for each task. The RA stage then generates a chromosome structure for the Mapping and Scheduling (MS) stage. The MS stage attempts to find an allocation for each task onto a core that maximizes the average QoS across all modes using the four mode response time analysis.

The chromosome in the RA stage has one integer gene for each task rep-

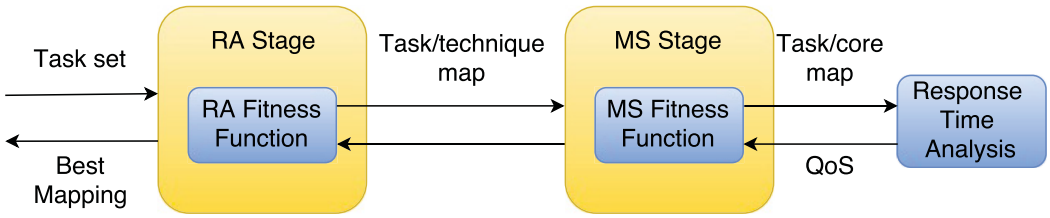


Fig. 5.3: DSE workflow using nested genetic algorithms.

resenting a fault tolerance mechanism. For instance, consider a task set with two HI tasks τ_1, τ_2 being mapped onto a platform that supports LS, DMR, and TMR. The chromosome would consist of two genes each limited to integers in the range $[0, 2]$.

The RA stage Fitness Function (RAFF) must determine the fitness (QoS) for each configuration of fault tolerance mechanisms. The FF creates a new task set using the transformations in Table 5.4 as well as the necessary constraints. The FF then creates a chromosome template for the MS stage based on the transformed task set. It is important that the task and replicas are represented by a single gene in the MS stage or else most chromosomes will result in illegal configurations after mutation and crossover.

Given a list of candidate cores that a task can be mapped to, it is possible to determine for each fault tolerance mechanism a mapping rule that generates a unique configuration from a random integer. Let m be the number of candidate cores and x be this random integer. Table 5.5 shows the number of possible configurations (possible task to core mappings) and the conversion rule for each type of mechanism. The conversion rule is a tuple with one entry per replica. For each replica, the conversion rule provides an index into the ordered list of candidate cores. The replica is mapped onto the core with this index.

The core is then removed from the list once it is allocated.

For example, consider a task and two replicas using TMR in a system with 5 processing cores. All three tasks must go on different cores. The number of configurations is $5 \cdot 4 \cdot 3 = 60$. The GA will generate a random integer in the range $[0, 59]$ representing a unique mapping of the three tasks onto the system, say 47. The number 47 is converted using the TMR rule to $(47/(4 \cdot 3), (47 \bmod (4 \cdot 3))/3, 47 \bmod 3) = (3, 3, 1)$. Suppose the core list is $\{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}$. The first copy is allocated to π_3 and π_3 is then removed from the list. The next copy is assigned to π_4 (now at index 3) and the third copy is assigned to π_1.

Table 5.5: Rules for generating unique MS configurations from an integer x for n cores

Technique	# Configurations	Conversion Rule
none	m	(x)
LS	m	(x)
DMR	$m(m-1)$	$(\frac{x}{m-1}, x \bmod (m-1))$
TMR	$m(m-1)(m-2)$	$(\frac{x}{(m-1)(m-2)}, \frac{x \bmod ((m-1)(m-2))}{m-2}, x \bmod (m-2))$
PR	$m^2(m-1)$	$(\frac{x}{m(m-1)}, \frac{x \bmod (m(m-1))}{m-1}, x \bmod (m-1))$

A unique MS stage is instantiated for each chromosome in the RA stage population. The MS stage generates a population based on the chromosome built by the RAFF. The Mapping and Scheduling stage Fitness Function (MSFF) builds each chromosome into a schedule and passes it along to the schedulability analysis. If the system is schedulable then the chromosome is assigned a fitness value equal to the average QoS across all four modes (defined as percentage of LO tasks that have not been dropped). If the analysis fails

then the chromosome is assigned a fitness value of 0.

It is possible to decouple the two stages of the genetic algorithm, for example to use a different DSE approach to perform the RA stage. However, the MS stage relies on the RA stage to prepare the chromosome structure so the new RA stage will need to provide this.

5.7 Experimental Results: Transient Faults

We conducted several experiments to compare the proposed model with the two-mode model and also to evaluate the DSE approach with ODR.

For all experiments, the periods of tasks were randomly selected from the set {10, 20, 40, 50, 100, 200, 400, 500, 1000} ms with uniform distribution. The utilization of each task in LO mode was generated using the UUnifast algorithm [101], such that the total LO-mode utilization across all tasks meets a given target, on average 80% for a pair of lockstep cores by default. We then calculated $C_i(LO) = u_i(LO).T_i$. For HI-criticality tasks, $CFactor = C(HI)/C(LO)$ is then randomly selected within the range $[1, 2]$ with uniform distribution, and $C(HI)$ was calculated as $C(HI) = CFactor \cdot C(LO)$. By default, half of the tasks were HI-critical. The HI mode PFH was set to 1×10^{-9}, equivalent to the avionics safety standard DO-178C level-A requirements. We assumed $\lambda_p = 1 \times 10^{-4}$. 1000 systems were generated for each combination of resource utilization, fraction of HI-criticality task utilization, and task count. We will first discuss the experiments performed on single core architectures then present the experiments utilizing our DSE approach on multicore architectures.

5.7.1 Experiments on single core architectures

First, we compare the LO-criticality task QoS (the fraction of LO tasks scheduled in a given mode) achieved by our four-mode model with the traditional two-mode model in a variety of scenarios. We varied system utilization, the total number of tasks, and the fraction of HI tasks.

Figure 5.4 shows the performance of each mode as utilization is increased for a single-core architecture. 20 tasks were scheduled, half of which are HI. We observe that in general, LO task QoS is better in OV and TF than HI: on average OV and TF execute 42.9% and 20.2% more LO-criticality tasks than HI respectively. The benefit is relatively small at low processor utilization: CPU idle time is available to execute longer tasks (OV mode) and task re-executions (TF mode), or even a combination (HI mode). As utilization increases, the QoS in HI mode degrades as more LO-criticality tasks must be dropped to accommodate HI-criticality tasks.

Figure 5.5 shows the performance of each mode as the percentage of HI tasks is increased from 20% to 70% while utilization is maintained at 80%. All other parameters are similar to the ones in Figure 5.4. Modes OV and TF maintain better QoS than HI throughout the experiment. The benefit of these two modes becomes higher as the percentage of HI tasks in the system increases, hence requiring more CPU time for overruns and re-executions.

Figure 5.6 shows the average improvement for modes OV and TF compared to HI as a function of the maximum number of faults (F) considered. When fewer faults are considered, the HI mode performs better and the relative improvement is lower: less time is needed for task re-execution, freeing more

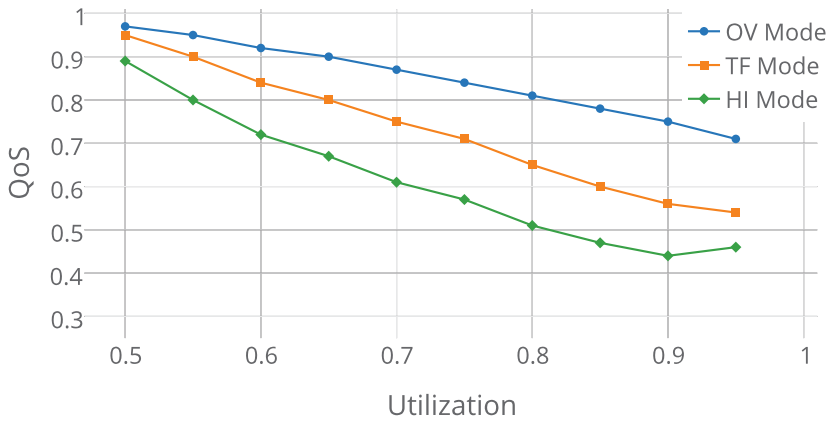


Fig. 5.4: Modes OV and TF achieve better QoS than HI for all utilizations (F not bounded).

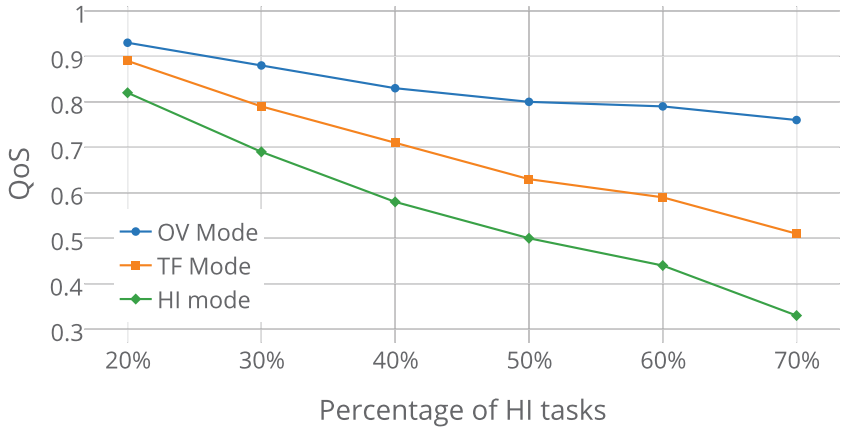


Fig. 5.5: Modes OV and TF achieve better QoS than HI for different percentages of HI tasks (F not bounded).

resources for LO-criticality tasks in HI mode. Also, limiting pessimism on the analysis is an effective way to improve QoS. For example, in mode TF (Figure 5.7), when $F = 2$, a 20.2% improvement in QoS is achieved by using the analysis in Section 5.4.1.

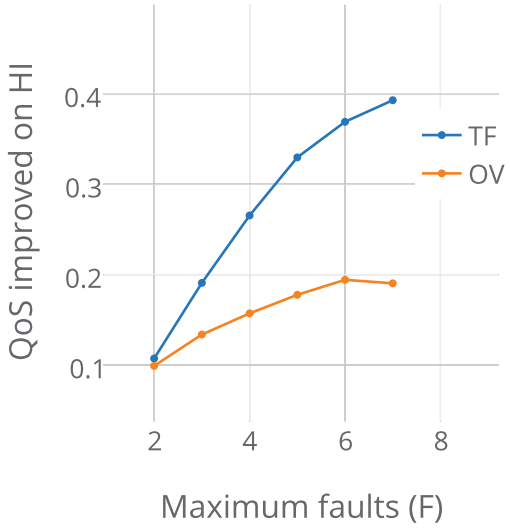


Fig. 5.6: Average improvement over all system utilizations for OV and TF modes compared to HI mode.

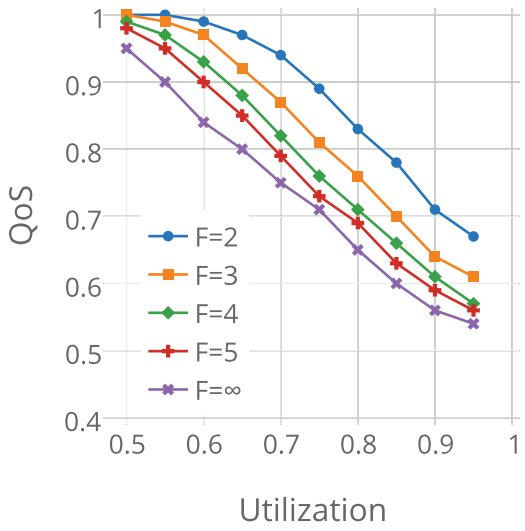


Fig. 5.7: Performance of TF mode for different F.

5.7.2 Experiments on multicore architectures

For multicore architectures, We conducted experiments to compare the LO tasks QoS achieved for various platform configurations. We first examined four

core systems with: (a) two pairs of lockstep (LS) cores, (b) one lockstep pair and one ODR pair (MIX), and (c) two ODR pairs (ODR) with only DMR. Additionally, we examined schedulability and QoS for the ODR system with DMR compared to a system capable of DMR, TMR, and PR. We varied system utilization and the fraction of HI tasks.

Fig. 5.8 compares the QoS for the different configurations as system utilization varies. 20 tasks were scheduled with half on average HI. The QoS for MIX and ODR is on average 20% better than for LS and 30% in the worst case. Fig. 5.9 shows similar results with the percentage of HI tasks varied while utilization is held constant at 0.7. In both cases, we observe that ODR provides better resource utilization on average as demand for the core increases with an overall average improvement of 29.8% for MIX and 28.5% for ODR. These figures indicate that combining ODR with MCS design allows providing more service to non-critical tasks without sacrificing reliability guarantees provided to critical ones. The results do not take into account possible voting overheads or recovery delays that might further differentiate the MIX and ODR platforms.

Figs. 5.10 and 5.11 compare the QoS and schedulability, respectively, for two ODR platforms. The first is only capable of DMR while the second is capable of all three varieties of ODR studied (DMR+TMR+PR). We observe that in general having more varieties of ODR improves both schedulability (8.3% improvement on average) and QoS (25% improvement on average). The different techniques provide greater flexibility in how to distribute the workload across the cores. This leads to an increase in schedulability, which allows us to

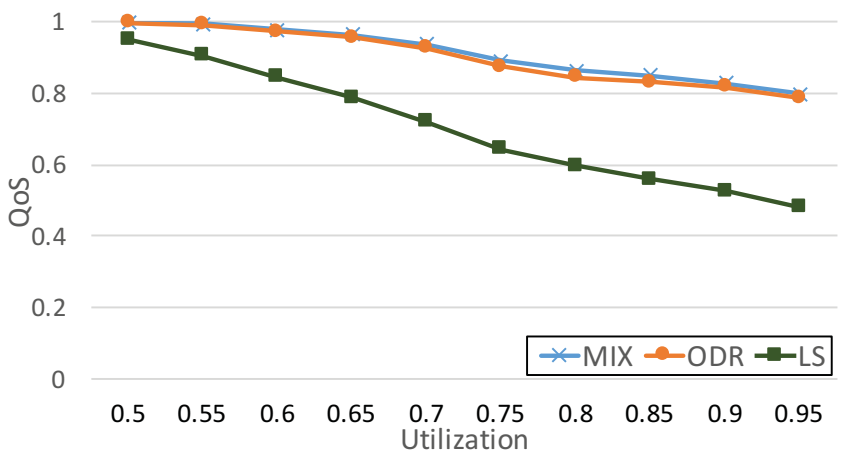


Fig. 5.8: ODR provides better QoS in multicore systems as utilization increases.

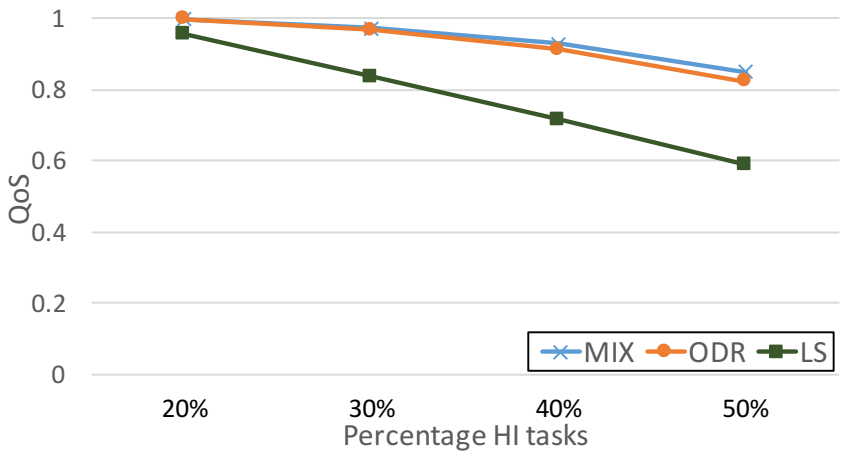


Fig. 5.9: ODR provides better QoS in multicore systems as the percentage of HI tasks increases.

utilize the available processing power in a more efficient manner, thus saving hardware resources while still maintaining safety when transient faults occur.

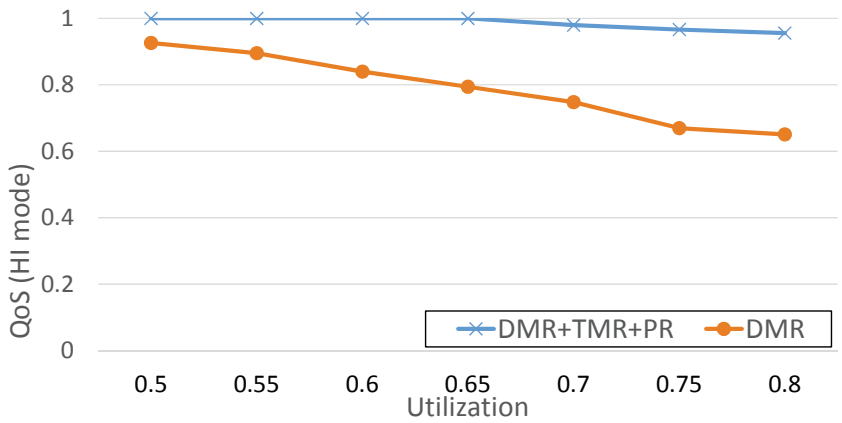


Fig. 5.10: Combining ODR techniques improves QoS.

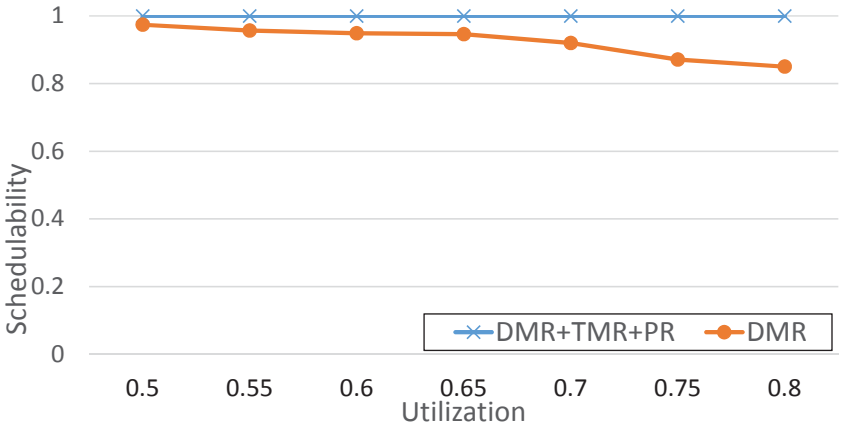


Fig. 5.11: Combining ODR techniques improves schedulability.

5.8 Scheduling MCS with Permanent Faults: Motivation

Most of the work on MCS has largely focused on execution time and, lately, transient faults. However, implementing MCS that are resilient to permanent faults has largely remained unaddressed [104]. In the remaining part of this chapter, we address this issue by presenting a design approach for MCS scheduled under permanent processor failures.

In MCS, the system copes with task overruns (which can be considered a type of fault) by moving into a new mode and dropping the non-critical tasks. We apply the same concept to deal with both processor failures and task overruns. Processor failures are tolerated by migrating the critical tasks from the failed processor to other working processors and dropping non-critical functions. For this purpose, we extend the MCS model and analysis to rake into consideration the case where one of the processors in the system fails.

While previous works targeting permanent faults in MCS (discussed in Section 2.5.3) focus on proposing reactive solutions to the case of a processor failure, our objective here is to consider the possibility of a processor failing when making the initial system design decisions. By incorporating task migration in design process, the reliability of the system is increased and the system is guaranteed to survive the failure of any single processor. Experiments show that this technique is able to schedule more systems than a 2-step process where migration is considered after the initial mapping is derived, and hence the proposed approach requires less resources to achieve similar reliability. To perform DSE, we propose a MILP-based process. The MILP finds (a) task allocations, (b) priority assignments, and (c) alternative assignments for each HI-criticality task if the core to which it is initially assigned fails.

5.9 System Operation with Permanent Faults

System operation proceeds as follows:

1. The system starts its operation by default in the LO mode. With all processors functional, each task executes on its original designated core.

All tasks, whether LO or HI, are allowed to execute. The task with the highest priority, regardless of its criticality level is executed.

2. While in LO mode, if any task executes beyond its $C(LO)$, the system moves into the HI mode. Under the HI mode, only HI tasks can execute and all LO tasks are dropped. Consistent with the previous works on MCS, we assume all processors enter the HI mode at the same time when a task executes beyond its $C(LO)$.

3. At any point and regardless of the system's mode, if a processor fails, then all of its HI-criticality tasks are distributed among working processors. If the system is in LO mode when the failure occurs, functional processors move immediately into the HI mode, dropping all LO-criticality tasks. Regardless of the system mode, once a processor failure is detected, its tasks migrate to predefined backup processors. Each working processor starts executing in the HI mode with a task set consisting of its own HI tasks and a subset of the failed processor's HI tasks.

The system must be able to accommodate all critical tasks even when a processors fails. A maximum of one failure is tolerated. Task allocations are determined offline. Each HI-criticality task has a *primary* processor on which it executes under normal scenarios and a *backup* processor on which it executes when the primary one fails. LO-criticality tasks only have a primary processor. We denote the primary processor for any task τ_i by a_i and the backup processor for any HI-criticality tasks τ_j by b_j.

Table 5.6: An Example Task Set

	L_i	a_i	b_i	π_i
τ_1	HI	p_1	p_2	7
τ_2	HI	p_1	p_2	6
τ_3	HI	p_2	p_1	5
τ_4	HI	p_2	p_3	4
τ_5	HI	p_3	p_1	3
τ_6	LO	p_1	-	2
τ_7	LO	p_2	-	1

5.10 Schedulability Analysis with Permanent Faults

The schedulability conditions are derived by extending the AMC-rtb [46] analysis. As described in Section 5.9, when a processor fails, its tasks are migrated to other processors. These processors drop their LO-criticality tasks and continue to execute both their HI tasks in addition to the migrating HI tasks from the failed processor. Schedulability analysis must take into account the requirements of these additional tasks. We will illustrate the different failures scenarios with the following example:

Assume the task set given in Figure 5.6 is allocated to three processors as shown in Figure 5.12. We will center the discussion around the lowest priority task τ_1. We need to make sure that τ_1 is schedulable when any processor in the system fails:

- If p_1 fails, tasks τ_1 and τ_2 will migrate to p_2 (depicted with black arrows in Figure 5.12. The LO-criticality task τ_6 is dropped. p_2 detects that p_1 failed, drops all LO tasks (in this case τ_7) and moves into the HI mode as described in Section 5.9. Now the task set running on p_2 consists of

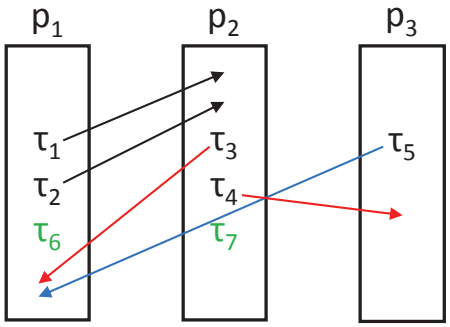


Fig. 5.12: An example system to illustrate the different scenarios to be considered in the schedulability analysis with failures.

four tasks (τ_1,τ_2,τ_3,τ_4). τ_1 can be preempted by any of the other three but it cannot be preempted by τ_7 since τ_7 was dropped at the same time that τ_1 migrated.

- If p_2 fails, task τ_3 will migrate to p_1 and τ_4 will migrate to p_3 (red arrows in Figure 5.12). Processor p_1 moves into the HI mode and drops τ_6. Tasks τ_2 and τ_3 can interfere with τ_1's execution. Furthermore, in the worst case τ_6 can delay τ_1's execution before the mode change. A safe upper bound on τ_1's response time can be obtained in a similar way to the regular mode change analysis since the mode change must occur before $R_1(LO)$ (otherwise τ_1's job would have finished), thus capping the maximum interference of τ_6 at $R_1(LO)$.

- If p_3 fails, tasks τ_5 migrates to p_1 (blue arrow in Figure 5.12). p_1 moves into HI and drops τ_6. This is similar to the previous case and mode change analysis is applicable.

Generalizing from this example, five cases must be checked for each task:

1. *No failures, LO*: verify the schedulability of all tasks in LO mode on their primary processors (Eq.(2.16)).

2. *No failures, MC*: verify the schedulability of HI tasks on their primary processors in the LO to HI mode change (Eq.(2.18)). This is subsumed by case 5 below.

3. *Primary processor failure*: verify the schedulability of each migrating HI task on its backup processor in the HI mode (the system moved into the HI mode when the failure occurred). LO tasks are dropped at the same time as migration and cannot affect the migrating task. The response time in this case is:

$$R_i^2(HI) = C_i(HI) + \sum_{j \in hp2H(i)} \left\lceil \frac{R_i^2(HI)}{T_j} \right\rceil \cdot C_j(HI) \tag{5.12}$$

   where $R_i^2(HI)$ denotes the HI mode response time of task τ_i after migrating to its backup processor. $hp2H(i)$ is the set of HI-criticality high priority tasks on τ_i's backup processor including those that migrated with τ_i. I.e., $hp2H(i)$=$\{j$: $(a_j = b_i \vee (a_j = a_i \wedge b_j = b_i)) \wedge \pi_j < \pi_i \wedge L_j = HI\}$.

4. *Other processor failure, HI*: verify the schedulability of HI tasks on their primary processors in the HI mode after migrations caused by a failure in another core. The response time must be checked for all $L-1$ one-core failure scenarios and is given by:

$$\forall p_x \neq a_i, \quad R_{i,p_x}^3(HI) = C_i(HI) + \sum_{j \in hp3H(i,x)} \left\lceil \frac{R_{i,p_x}^3(HI)}{T_j} \right\rceil \cdot C_j(HI) \tag{5.13}$$

where $R^3_{i,p_x}(HI)$ is the HI mode response time of task τ_i on its primary processor when processor p_x ($p_x \neq a_i$) fails. $hp3H(i,x)$ is the set of HI-criticality high priority tasks on τ_i's primary processor including those that migrated from p_x. i.e. $hp3H(i,x)$=$\{j$: $(a_j = a_i \vee (a_j = p_x \wedge b_j = a_i)) \wedge \pi_j < \pi_i \wedge L_j = HI\}$. This case is subsumed by case 5.

5. *Other processor failure, MC*: verify the schedulability of HI tasks on their primary processors in the LO to HI mode change caused by task migration from each other processor. The response time in this case is given by:

$$\begin{aligned} \forall p_x \neq a_i, \quad R^3_{i,p_x}(MC) &= C_i(HI) \\ &+ \sum_{k \in hp1L(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil \cdot C_k(LO) \\ &+ \sum_{j \in hp3H(i,x)} \left\lceil \frac{R^3_{i,p_x}(MC)}{T_j} \right\rceil \cdot C_j(HI) \end{aligned} \tag{5.14}$$

Case 5 needs to be checked with p_x set to each of the other L-1 processors in the system (excluding τ_i's primary processor). Eq.(5.14) provides a safe upper bound regardless of the scenario. A mode change caused by a processor failure is a worse scenario than a mode change caused by a task overrun as the latter does not introduce new tasks. The worst case scenario occurs when a processor failure causing a mode change occurs immediately after all LO-criticality high priority jobs finish their execution. For this scenario, the maximum possible contribution for the LO tasks is considered in the second term in Eq.(5.14). The additional computation required by the migrating tasks is considered as part of the third term.

It is clear that case 5 subsumes 4 since Eq.(5.14) has an additional term. Moreover case 5 subsumes 2 since Eq.(2.18) uses the set $hpH(i)$ which is a subset of $hp3H(i,x)$. In practice, we need to check cases 1 (Eq.(2.16)), and 3 (Eq.(5.12)) once, and case 5 (Eq.(5.14)) L-1 times.

5.11 MILP-Based DSE with Permanent Faults

In this section, we show how a mixed-criticality task set can be implemented on a multicore platform such that it remains schedulable even under a processor failure. Three design variables can be controlled for each task: primary allocation, backup allocation (for HI tasks), and priority. The remaining task parameters ($C(LO)$,$C(HI)$, T, D) are considered constants in the formulation. The main design constraints are the scheduling constraints given by Eqs.(2.16), (5.12), and (5.14). All the MILP variables have non-negativity constraints.

5.11.1 General system variables and constrains

The allocation of tasks to their primary processors is represented by the binary variable ($A_{i,x}$). This variable is set to 1 if processor p_x is τ_i's primary processor and 0 otherwise. Each task has exactly one primary processor. This is expressed by the following constraint:

$$\forall \tau_i : \sum_{x=1}^{L} A_{i,x} = 1 \tag{5.15}$$

Similarly, we define the binary variable $B_{i,x}$ for backup allocations. $B_{i,x}$ is set to 1 if processor p_x is HI-criticality task τ_i's backup processor and 0

otherwise. For LO-criticality tasks, $B_{i,x}$ is set to 0.

$$\begin{gathered}\forall \tau_i \in \Gamma_{HI} : \sum_{x=1}^{L} B_{i,x} = 1 \\ \forall \tau_i \in \Gamma_{LO} : B_{i,x} = 0\end{gathered} \tag{5.16}$$

The task's primary processor and backup processor must be different. This is expressed by the following constraint:

$$\forall \tau_i, \forall p_x : A_{i,x} + B_{i,x} = 1 \tag{5.17}$$

For defining scheduling constraints, it is important to recognize tasks allocated to the same core. For this, a binary variable $G_{i,j}$ is defined and set to 1 if tasks τ_i and τ_j are assigned to the same primary processor and 0 otherwise. This is expressed by the following two constraints:

$$\forall \tau_i, \tau_j, \tau_i \neq \tau_j, \forall p_x : G_{i,j} \geq A_{i,x} + A_{j,x} - 1 \tag{5.18}$$

$$\begin{gathered}\forall \tau_i, \tau_j, \tau_i \neq \tau_j, \forall p_x, p_y, p_x \neq p_y : \\ G_{i,j} \leq 2 - A_{i,p_x} - A_{j,p_y}\end{gathered} \tag{5.19}$$

The first constraint ensures that $G_{i,j}$ is set to 1 if they are on to the same primary processor p_x, and the second ensures it is set to 0 if they are on to different ones (p_x and p_y).

When failures occur, task allocations change. For this, another binary variable $H_{i,j}$ is defined. $H_{i,j}$ is set to 1 if the HI-criticality tasks τ_i and τ_j are allocated to the same core when τ_i's processor fails. This happens either if (a) τ_i's backup processor is τ_j's primary (Eq.(5.20)) or (b) τ_i and τ_j migrate together and hence have the same primary and backup processors (Eq.(5.21)). Otherwise, the constraints in Eqs.(5.22) and (5.23) ensure $H_{i,j}$ is 0.

$$\forall \tau_i, \tau_j \in \Gamma_{HI}, \tau_i \neq \tau_j, \forall p_x : H_{i,j} \geq B_{i,x} + A_{j,x} - 1 \tag{5.20}$$

$$\forall \tau_i, \tau_j \in \Gamma_{HI}, \tau_i \neq \tau_j, \forall p_x : H_{i,j} \geq G_{i,j} + B_{i,x} + B_{j,x} - 2 \tag{5.21}$$

$$\begin{gathered} \forall \tau_i, \tau_j \in \Gamma_{HI}, \tau_i \neq \tau_j, \forall p_x, p_y, p_x \neq p_y : \\ H_{i,j} \leq 2 - B_{i,p_x} - A_{j,p_y} + G_{i,j} \end{gathered} \tag{5.22}$$

$$\begin{gathered} \forall \tau_i, \tau_j \in \Gamma_{HI}, \tau_i \neq \tau_j, \forall p_x, p_y, p_x \neq p_y : \\ H_{i,j} \leq 3 - B_{i,p_x} - B_{j,p_y} - G_{i,j} \end{gathered} \tag{5.23}$$

Task priority is specified by another binary variable, $\pi_{i,j}$. This variable is 1 if task τ_i has higher priority than task τ_j and 0 otherwise. To ensure the correctness of this variable, antisymmetry and transitivity properties must be enforced on π. This is done with the following two constraints:

$$\forall \tau_i, \tau_j, \tau_i \neq \tau_j : \pi_{i,j} + \pi_{j,i} = 1 \tag{5.24}$$

$$\begin{gathered} \forall \tau_i, \tau_j, \tau_k, \tau_i \neq \tau_j \neq \tau_k : \\ \pi_{i,k} \geq \pi_{i,j} + \pi_{j,k} - 1 \end{gathered} \tag{5.25}$$

5.11.2 Schedulability constrains

A system is schedulable if the response times calculated using Eqs.(2.16), (5.12), and (5.14) are all less than the task's deadline. These response times are represented by the ILP variables R_i^1, R_i^2 and the array $R_{i,x}^3$ respectively. The system is schedulable if the following constraints are satisfied:

$$\forall \tau_i : R_i^1 \leq D_i \tag{5.26}$$

$$\forall \tau_i : R_i^2 \leq D_i \tag{5.27}$$

$$\forall \tau_i, \forall p_x : R_{i,x}^3 \leq D_i \tag{5.28}$$

To formulate Eq.(2.16) we need to linearize the ceiling function that represents the number of preemptions by high priority tasks that have the same primary processor as the task τ_i. We define an integer variable $P^1_{i,h}$ representing the number of such preemptions caused by τ_h. $P^1_{i,h}$ is bounded by:

$$\forall \tau_i, \tau_h, \tau_i \neq \tau_h : P^1_{i,h} \geq \frac{R^1_i}{T_h} - M * (1 - G_{i,h}) - M * \pi_{i,h} \tag{5.29}$$

$$\forall \tau_i, \tau_h, \tau_i \neq \tau_h : P^1_{i,h} \leq 1 - \epsilon + \frac{R^1_i}{T_h} \tag{5.30}$$

where ϵ is a very small value and M is a large integer constant. The constraint in Eq.(5.29) is disabled whenever τ_h has a different primary processor ($G_{i,h}$=0) or has a lower priority than τ_i ($\pi_{i,h}$=1). This ensures that only tasks in the set $hp(i)$ are accounted for. The constraint in Eq.(5.30) ensures that the left side value does not exceed $\frac{R^1_i}{T_h}$ by more than 1. The response time R^1_i is then given by:

$$\forall \tau_i : R^1_i = C_i(LO) + \sum_{h=1, h\neq i}^{N} C_h(LO).P^1_{i,h} \tag{5.31}$$

In a similar way, we formulate Eq.(5.12). We define $P^2_{i,h}$ to be the number of preemptions by a high priority HI-criticality task τ_h that executes on the same core as τ_i when τ_i's primary fails (the set $hp2H(i)$). $P^2_{i,h}$ is bounded by:

$$\forall \tau_i, \tau_h, \tau_i \neq \tau_h, \tau_h \in \Gamma_{HI} : P^2_{i,h} \geq \frac{R^2_i}{T_h} - M * (1 - H_{i,h}) - M * \pi_{i,h} \tag{5.32}$$

$$\forall \tau_i, \tau_h, \tau_i \neq \tau_h, \tau_h \in \Gamma_{HI} : P^2_{i,h} \leq 1 - \epsilon + \frac{R^2_i}{T_h} \tag{5.33}$$

$H_{i,h}$ as mentioned earlier is 1 when τ_i and τ_h are allocated to the same core if τ_i's primary processor fails, hence the constraint in Eq.(5.32) is disabled

whenever τ_h is not in the set $hp2H(i)$. The response time R_i^2 is then given by:

$$\begin{aligned} \forall \tau_i \in \Gamma_{HI} : R_i^2 = C_i(HI) + \sum_{h=1, h\neq i}^{N} C_h(HI).P_{i,h}^2 \\ \forall \tau_i \in \Gamma_{LO} : R_i^2 = 0 \end{aligned} \tag{5.34}$$

Finally, we need to check if a HI task is schedulable on its primary processor when another processor fails (Eq.(5.14)). We define a variable $P_{i,h,x}^3$ to denote the number of preemptions by a high priority HI-criticality task τ_h that executes on the same processor as τ_i when processor p_x fails (the set $hp3H(i,x)$). $P_{i,h,x}^3$ is constrained by the following:

$$\begin{aligned} \forall \tau_i, \tau_h, \tau_i \neq \tau_h, \tau_h \in \Gamma_{HI}, \forall p_x : P_{i,h,x}^3 \geq \frac{R_{i,x}^3}{T_h} \\ -M * (1 - H_{h,i}) - M * \pi_{i,h} - M * A_{i,x} - M * (1 - A_{h,x}) \end{aligned} \tag{5.35}$$

$$\begin{aligned} \forall \tau_i, \tau_h, \tau_i \neq \tau_h, \tau_h \in \Gamma_{HI}, \forall p_x : P_{i,h,x}^3 \geq \frac{R_{i,x}^3}{T_h} \\ -M * (1 - G_{i,h}) - M * \pi_{i,h} - M * A_{i,x} \end{aligned} \tag{5.36}$$

$$\forall \tau_i, \tau_h, \tau_i \neq \tau_h, \tau_h \in \Gamma_{HI}, \forall p_x : P_{i,h,x}^3 \leq 1 - \epsilon + \frac{R_{i,x}^3}{T_h} \tag{5.37}$$

The first constraints ensures that $P_{i,h,x}^3$ accounts for preemptions by any migrating HI task τ_h that: (1) is allocated initially to the failing core, p_x ($A_{h,x} = 1$), (2) moves to the same core as τ_i when its core fails ($H_{h,i}$=1), and (3) has higher priority ($\pi_{i,h}$=0). The term $M * A_{i,x}$ is used in both Eq.(5.35) and Eq.(5.36) and ensures disabling the constraints if p_x is τ_i's primary and hence $P_{i,h,x}^3$ is 0 when $a_i = p_x$. The constraint in (Eq.(5.36)) ensures that $P_{i,h,x}^3$ accounts for preemptions by HI tasks that: (1) are allocated to τ_i's core ($G_{i,h}$=1), and (2) have higher priority ($\pi_{i,h}$=0). With these constraints, the

set $hp3H(i,x)$ is fully accounted for. The third constraint places an upper bound on $P^3_{i,h,x}$. Finally $R^3_{i,x}$ is given by:

$$\begin{aligned} \forall \tau_i \in \Gamma_{HI}, \forall p_x : R^3_{i,x} = C_i(HI) + \sum_{h=1,h\neq i}^{N} C_h(HI).P^3_{i,h,x} \\ \forall \tau_i \in \Gamma_{LO}, \forall p_x : R^3_{i,x} = 0 \end{aligned} \tag{5.38}$$

The values of the MILP variables A, B, and π respecting these constraints represent a system configuration that is both schedulable and tolerant of any single processor failure.

5.12 Experimental Results: Permanent Faults

We conducted experiments to evaluate the performance of our proposed approach on MCS task sets. Since no similar relocation approach exists for the the standard MCS model, we compared our work with a baseline 2-step design approach. In this approach, an MILP formulation is first used to find an initial allocation assuming no failures. Based on this allocation, another formulation is used to find backup processors for failing HI tasks. HI tasks on the working processors remain while HI tasks on the failed core migrate. The formulations were implemented in CPLEX [92] and run on servers equipped with Intel Xeon X5650 processors running at 2.67 GHz and with 4GBs of memory. CPLEX was set to time out after two hours if no feasible solution had been found.

We generated task sets with the following parameters: The periods were randomly selected from the set {10, 20, 40, 50, 100, 200, 400, 500, 1000} ms

with uniform distribution. The utilization $u_i(LO)$ of each task in the LO mode was generated using the UUnifast algorithm [101] such that the total LO mode utilization meets a given target. We set the LO mode WCET for any task τ_i to $C_i(LO) = u_i(LO)/T_i$. For HI tasks, the HI mode WCET ($C_i(HI)$) was then randomly selected in the range [$C_i(LO)$,3$C_i(LO)$] with uniform distribution. By default, half of the tasks were HI-criticality. The number of tasks and total LO mode utilization were varied on a platform consisting of four homogeneous cores. 100 systems were generated for each parameter configuration.

Figure 5.13 shows the percentage of systems that are both schedulable and can survive a processor failure at different LO-mode utilizations for both the proposed approach and the baseline 2-step approach. For this experiment 20 tasks were scheduled on 4 cores. The proposed approach achieves a 3.2X increase in the number of schedulable systems compared to the 2-step process over the utilization range. Another way to look at it is that the proposed approach can be used to ensure reliability through proper system configuration. Achieving similar reliability against processor failures would otherwise require additional resources. A similar result is obtained for systems composed of 3 cores and 12 tasks (Figure 5.14).

We also performed an experiment to observe the behavior of the ILP as the size of the task set changes. Figure 5.15 shows the percentage of: schedulable, unschedulable, and timed out systems using the proposed formulation and the 2-step formulation. Systems consisting of 10-40 tasks and an overall LO mode utilization of 0.7 were scheduled on 4 cores. The proposed approach schedules more systems at each size. For systems with 10-25 tasks, the number of systems

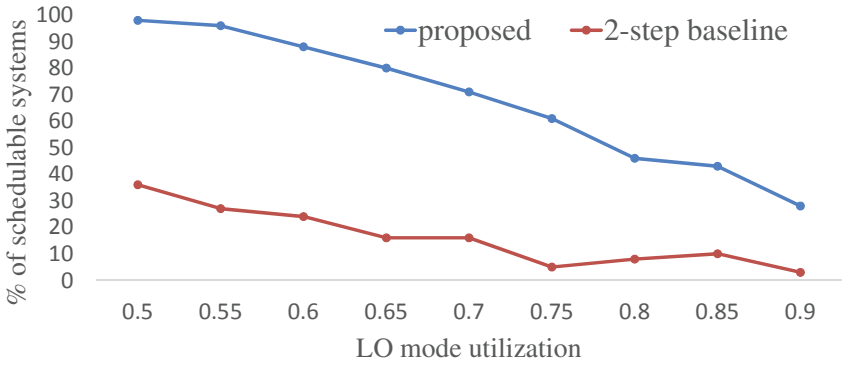


Fig. 5.13: Schedulability at different utilizations (N=20, L=4).

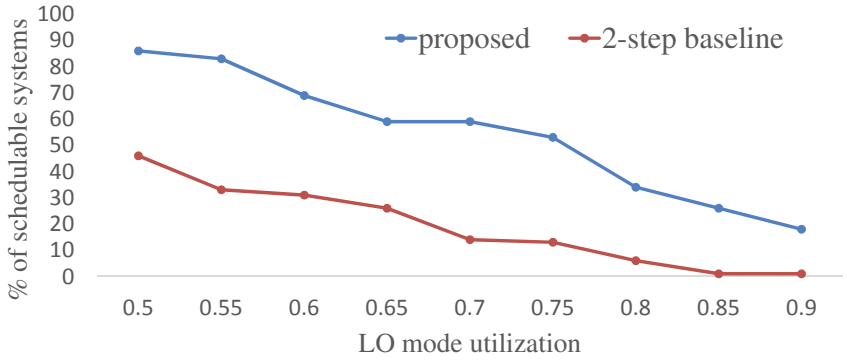


Fig. 5.14: Schedulability at different utilizations (N=12, L=3).

scheduled by the proposed approach increases with N as the average task size ($0.7^*L/N$) decreases and hence these smaller tasks allow more flexibility in scheduling. With $N \geq 30$, we see an increase in the number of systems that the solver cannot find a solution for within the 2-hour limit. For moderate size systems, this can be addressed by increasing this limit. The task allocation problem is known to be NP-hard, therefore, the solver might not find feasible solutions within an acceptable time for larger systems. Sub-optimal solutions can be found using heuristics which will be the subject of future research.

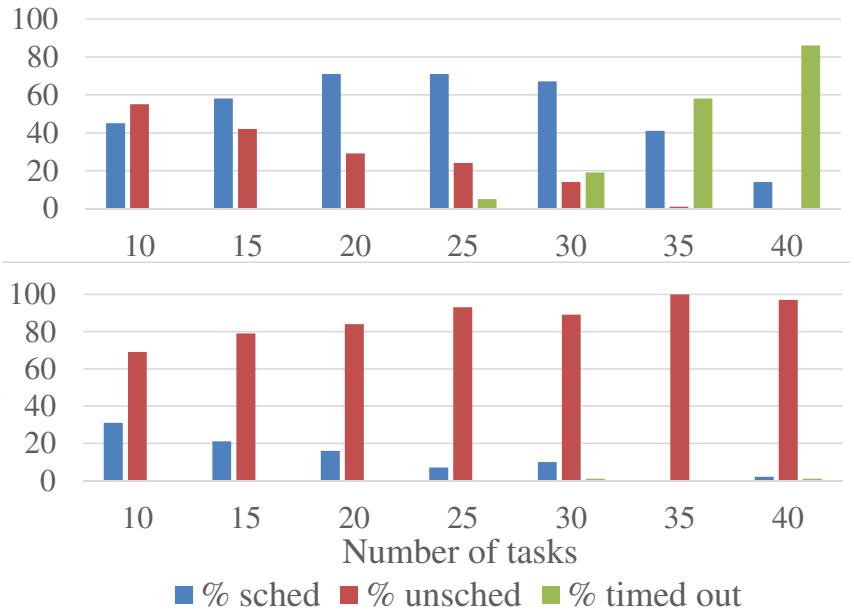


Fig. 5.15: Schedulability at different task set sizes ($U(LO)$=0.7, L=4): proposed (top), 2-step baseline (bottom).

5.13 Conclusion

In this chapter, we presented solutions for designing efficient and reliable MCS. First, to deal with transient faults, we proposed a four-mode model for fault-tolerant multi-core MCS. This model differentiates between overruns and faults, and also supports on-demand redundancy and heterogeneous systems, to improve LO-criticality task QoS without affecting the reliability and scheduling guarantees provided to HI-criticality tasks. A novel task set transformation approach facilitates an efficient implementation of nested genetic algorithms that perform redundancy technique assignment and task mapping. Supported by our analysis, design space exploration over heterogeneous hardware configurations and arbitrary ODR mechanisms demonstrated that in general, systems with at least partial support for ODR outperform static lockstep,

achieving 29% higher QoS on average. We further demonstrated that support for several types of ODR can further improve both the average schedulability and QoS.

For tolerating permanent faults, we extended the MCS model to support such failures and derived the new model's schedulability analysis. Moreover, we proposed a design space exploration approach that considers fault-tolerance an essential part of the design problem, thereby producing efficient and reliable systems. Experimental results show that our approach outperforms a baseline 2-step exploration process. For example, in systems composed of 20 tasks and 4 cores, a 3.2X improvement in schedulability is observed while maintaining guarantees of safe operation in case of any single processor failure.

Chapter 6

Implementing Synchronous Reactive Models of Mixed-Criticality Systems

In this Chapter [1], we turn to more practical problem faced by designers when designing complex Mixed-Criticality Systems (MCS). Complex embedded designs in many instances are designed using a model-based design process enabling early verification of functionality and automatic synthesis of implementation. The functionality of the system is specified using high level formal models such as Synchronous Reactive (SR) models. It is important in a model-based design process to preserve the semantics of the model in the generated implementation regardless of the platform and scheduling model used. To synthesize a correct, semantics-preserving implementation from an SR model, the

[1] The work in this chapter was done in close collaboration with Qingling Zhao. My contribution was being the main developer of the branch-and-bound based algorithm and phase 2 of the heuristic with input from Ms. Zhao. I also implemented the parts I contributed.

signal flow relationships among nodes in the ideal logical time execution must be preserved, regardless of possible preemptions or variable execution times. A multi-task implementation may not be schedulable unless the designer modifies the model by adding functional delays on selected edges, with costs in terms of degraded control performance, representing a tradeoff between schedulability and control performance.

In this chapter, we consider the problem of optimizing the implementation of mixed-criticality SR models with fixed-priority scheduling on a uniprocessor, with the following problem formulation: *find the schedulable implementation that requires addition of the minimum weighted sum of functional delays for a mixed-criticality SR model on a single processor platform with fixed priority scheduling.* We present an optimal algorithm based on Branch-and-Bound search, and an effective heuristic algorithm.

The rest of this chapter is organized as follows, In Section 6.1, we elaborate more on the schedulability vs functional delay tradeoff with a concrete example. In Section 6.2, we present the system model. In Section 6.3, we discuss the schedulability analysis. In Section 6.4, we discuss the priority assignment algorithm reused here. In Section 6.5, we present our two optimization algorithms. The experimental results are presented in Section 6.6. Finally Section 6.7 concludes the chapter.

6.1 Schedulability vs Functional Delay Tradeoff: Example

As the Simulink example in Section 2.6.1 showed, there is a cost to adding functional delays to communication links in an SR model. While some delays can be avoided, sometimes it is necessary to add functional delays to ensure semantics-preserving schedulable implementations. We will illustrate this with the example in Figure 6.1:

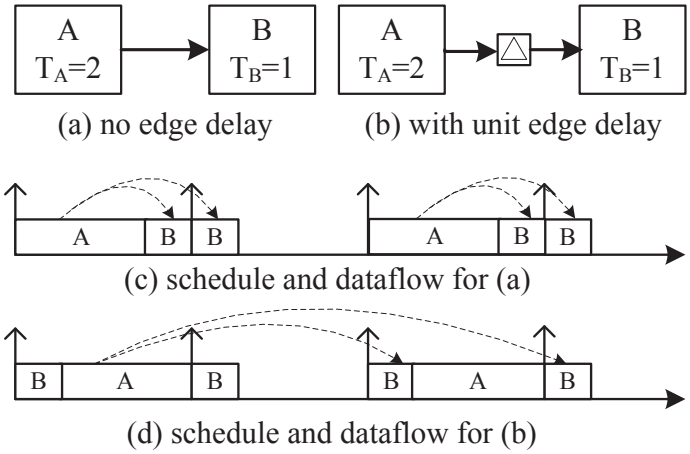


Fig. 6.1: Multirate SR models without edge delay, shown in (a), and with unit edge delay, shown in (b), and the corresponding schedules, shown in (c) and (d).

Figure 6.1 shows a multi-rate SR model with or without unit delay on a low-rate-to-high-rate dependency edge. Assuming each node is mapped to a task, the two tasks A and B are periodic, independent tasks, triggered by the OS clock tick. In (a), task A with period $T_A = 2$ produces output that is consumed by task B with period $T_B = 1$, without any delay on the edge between A and B. In the corresponding schedule in (c), there is a precedence constraint,

i.e., task A must execute before a simultaneously-released task B in order to ensure the correct dataflow from A to B, as denoted by the dotted arrows. This can be achieved by assigning A a higher priority than B. Since tasks A and B execute on the same processor and are driven by the same OS clock tick, the higher-priority task A is always placed ahead of a simultaneously-released task B in the OS ready queue, and executed before B. In (b), there is a unit delay on the edge between A and B. In the corresponding schedule in (d), there is an additional latency of one period of A ($T_A = 2$) for the dataflow from A to B, hence the precedence constraint from A to B is eliminated, so we can adopt the priority assignment that maximizes schedulability. It is well-known that the *rate monotonic* priority assignment is optimal for fixed-priority scheduling, where each task's priority is inversely proportional to its period. We adopt the rate monotonic priority assignment in (d) to maximize schedulability. Suppose A's Worst Case Execution Time (WCET) is increased slightly, then the schedule in (c) will incur deadline misses for B, while the schedule in (d) will continue to be schedulable, thanks to the optimality of rate-monotonic priority assignment. Note that tasks A and B are always periodic, independent tasks, and the correct inter-task communication semantics is guaranteed by priority assignment and/or delay insertion, not by explicit precedence constraints on task releases.

This example illustrates the tradeoff between adding edge delays for improved schedulability, and not adding edge delay for reduced end-to-end dataflow latency and improved control performance. If it is not possible to find a schedulable implementation, edge delays can be added to remove the precedence

constraints and improve schedulability, at the cost of degraded control performance. In this chapter, we propose algorithms targeting this tradeoff for MCS.

6.2 System Model

In this chapter, we use two MCS task models: the basic Adaptive Mixed-Criticality (AMC) model (described in Section 2.4.1, schedulability analysis in Section 2.4.2), and a modified version of the Elastic Mixed-Criticality (EMC) model presented in Section 4.1.

6.2.1 Task model

In a mixed-criticality SR model, the output nodes of the model (those generating output values used by actuators, or other nodes outside of the model) are designated with a criticality level by the designer based on application-level semantics. The criticality level is prorogated backwards from the output nodes, i.e., the criticality level of a non-output node is assigned HI if it is a predecessor of a HI-criticality node, or LO otherwise, i.e., a non-output node is HI-criticality if it is a predecessor of both HI-criticality and LO-criticality nodes. There is a many-to-one relationship between node and tasks, i.e., one or more nodes are grouped into a single task. In this work, we assume that the grouping has been performed, hence each node N_i in the SR model corresponds to a task τ_i, with the following parameters:

$$\langle L_i, T_i, D_i, C_i(LO), C_i(HI) \rangle$$

– L_i is its criticality level, HI or LO.

– T_i is its nominal period.

– D_i is its relative deadline. We assume the implicit deadline model ($D_i = T_i$).

– $C_i(LO)$ is its WCET in LO-criticality mode.

– $C_i(HI)$ is its WCET in HI-criticality mode. We assume that $C_i(LO) \leq C_i(HI)$ if $L_i = HI$; $C_i(LO) = C_i(HI)$ if $L_i = LO$.

6.2.2 EMC model with no mode change task dropping

For most applications, it is desirable to provide some degraded Quality of Service (QoS) to LO-criticality tasks in HI-criticality mode, instead of dropping them completely as in AMC. We define our EMC model for fixed-priority scheduling, by allowing a LO-criticality task τ_i to have a LO-criticality period $T_i(LO)$ equal to its nominal period, i.e., $T_i(LO) = T_i$; and a HI-criticality period $T_i(HI) \geq T_i(LO)$, reflecting a reduced QoS of LO-criticality tasks in HI-criticality mode. After mode switch to HI-crit, τ_i can still be executed but *with an extended period* $T_i(HI)$. The *period scaling factor* k_i is defined to be $k_i = T_i(HI)/T_i(LO) \geq 1$. A larger value of k_i implies lower QoS for LO-criticality tasks in HI-criticality mode, but better schedulability, hence k_i is a tunable parameter that controls the QoS vs. schedulability tradeoff. Each task τ_i's relative deadline is always kept to be equal to its period ($D_i = T_i$).

In Chapter 4, due to the nature of the problem considered (multicore scheduling with migrations in mode changes), we made the assumption that LO tasks are dropped during a mode change, then they start on their new cores when the mode change terminates. We target single core architectures

in this chapter and make a more conservative assumptions that LO tasks must be executed during a mode change. This requires changing the schedulability conditions. The modified schedulanbility analysis for EMC will be presented in Section 6.3.

6.2.3 System operation

For AMC, system operation proceeds as described in Section 2.4.1

The scheduling rules for EMC are similar to those for AMC, except the following change to step 4 in the system operation description in Section 2.4.1:

4- When this happens (a task executes beyond $C(LO)$), LO tasks are allowed to continue execution with an extended period $T_i(HI) \geq T_i(LO)$. HI tasks are allowed to execute upto their certification authority accepted WCET estimate (C(HI)).

Consider two special cases: if $k_i = 1$, then task τ_i keeps running with the same period $T_i(LO)$ in HI-criticality mode, so the task model is equivalent to the non-mixed-criticality regular task model; if $k_i = \infty$, then task τ_i is effectively dropped in HI-criticality mode, so the task model is equivalent to the AMC model. Note that the EMC model corresponds to the *Extended Elastic Mixed-Criticality* task model in [97], which uses EDF scheduling instead of fixed-priority scheduling as in this work. [93] also proposed to assign a pair of minimum and maximum periods to each LO-criticality task, but a LO-criticality task may execute with any period within $[T_i(LO), T_i(HI)]$ by exploiting runtime slack in the static schedule, without a distinct time instant of mode switch from LO-criticality to HI-criticality mode.

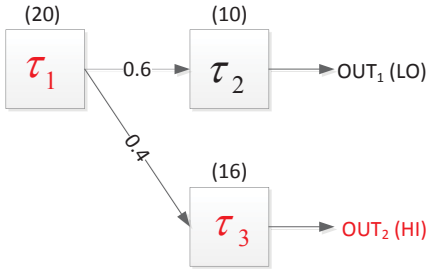


Fig. 6.2: An example mixed-criticality SR model. The number in parentheses above a block denotes its period, and the number along the edge denotes its cost.

Table 6.1: Timing parameters for the example in Figure 6.2.

τ_i	L_i	$T_i(LO)$	$T_i(HI)$	$C_i(LO)$	$C_i(HI)$	pri
τ_1	HI	20	20	3	4	3
τ_2	LO	10	20	4	4	2
τ_3	HI	16	16	4	5	1

Figure 6.2 shows an example of mixed-criticality multi-rate synchronous program, with timing parameters specified in Table 6.1. The output block τ_2 is LO-criticality and τ_3 is HI-criticality; the non-output blocks τ_1 is assigned with HI criticality since it is a predecessor of HI-criticality output block τ_3. Priority order from high to low is: τ_1, τ_2, τ_3.

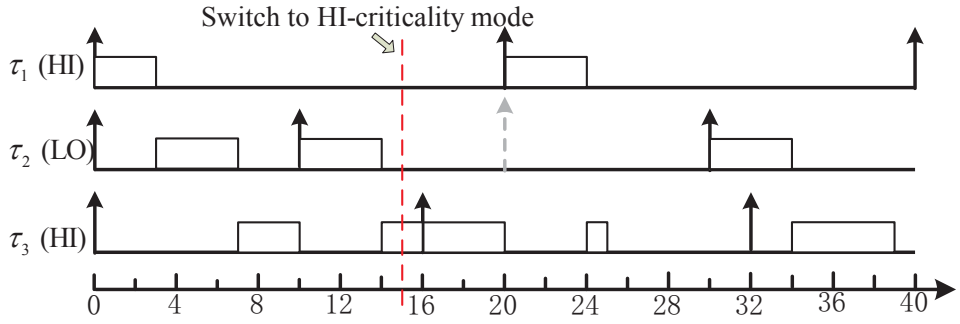


Fig. 6.3: An example of EMC.

Figure 6.3 shows a possible schedule of the EMC model, with a criticality

change from LO to HI-criticality mode at time 15 when task τ_3 exceeds its LO-criticality WCET $C_3(LO)$. Afterwards, jobs from LO-criticality task τ_2 will be released with extended period 20 instead of 10, so the third job of τ_2 is released at time 30 instead of 20.

We develop algorithms that intend to assign optimal values to two **design variables**: functional delay addition and priority assignment. The **optimization objective** is to minimize the sum of edge delays, where the edge weight is determined based on the effect of unit delay on the control performance.

6.3 Schedulability Analysis

Table 6.2 lists some notations used in this chapter.

Table 6.2: Notations used in this chapter

$hp(i)$	The set of (LO-criticality and HI-crit) tasks with higher priority than τ_i
$hpL(\tau_i)$	The set of LO-criticality tasks with higher priority than τ_i
$lpL(\tau_i)$	The set of LO-criticality tasks with lower priority than τ_i
$hpH(\tau_i)$	The set of HI-criticality tasks with higher priority than τ_i
$lpH(\tau_i)$	The set of HI-criticality tasks with lower priority than τ_i

Schedulability analysis for AMC was presented in Section 2.4.2. We now present our schedulability test for EMC, which is based on AMC-rtb [46].

In LO-criticality mode, there is no difference between EMC and AMC, hence τ_i's Worst Case Response Time (WCRT) in LO-criticality mode, $R_i(LO)$, can be computed with the same Equation (2.16).

Since LO-criticality tasks continue to execute in HI-criticality mode, we need to verify the schedulability of both HI-criticality tasks and LO-criticality

tasks in HI-criticality mode. In the steady HI-criticality mode, each HI-criticality task τ_j has WCET $C_j(HI)$ and period T_j, and each LO-criticality task τ_k has WCET $C_k(LO)$ and period $T_k(HI)$, hence task τ_i's WCRT in the steady HI-criticality mode is:

$$R_i(HI) = C_i(L_i) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil \cdot C_j(HI) + \sum_{k \in hpL(i)} \left\lceil \frac{R_i(HI)}{T_k(HI)} \right\rceil C_k(LO) \tag{6.1}$$

where $C_i(L_i)$ is τ_i's WCET at its own criticality level $L_i = LO \vee HI$.

For the Mode Switch (MS), we consider a crossing job J_i of task τ_i that is released in LO-criticality mode but finished in HI-criticality mode. The worst-case interference to J_i from each $\tau_j \in hpH(i)$ is the same as in AMC:

$$\left\lceil \frac{R_i(MS)}{T_j} \right\rceil \cdot C_j(HI) \tag{6.2}$$

A job of LO-criticality task $\tau_k \in hpL(i)$ that is both released and finished in LO-criticality mode has period $T_k(LO)$. The maximum number of such jobs of task $\tau_k \in hpL(i)$ that can cause interference to J_i is:

$$n_k^{LO} = \left\lfloor \frac{R_i(LO)}{T_k(LO)} \right\rfloor \tag{6.3}$$

A job of LO-criticality task $\tau_k \in hpL(i)$ that is either released in HI-criticality mode, or released in LO-criticality mode but finished in HI-criticality mode, has period $T_k(HI)$. The maximum number of such jobs of task $\tau_k \in$

$hpL(i)$ that can cause interference to J_i is:

$$\left\lceil \frac{R_i(MS) - n_k^{LO} \cdot T_k(LO)}{T_k(HI)} \right\rceil \tag{6.4}$$

Since a LO-criticality task τ_k has the same WCET of $C_k(LO)$ in both LO-criticality mode and HI-criticality mode, the worst-case interference to J_i from task $\tau_k \in hpL(i)$ is upper-bounded by:

$$\left(n_k^{LO} + \left\lceil \frac{R_i(MS) - n_k^{LO} \cdot T_k(LO)}{T_k(HI)} \right\rceil\right) \cdot C_k(LO) \tag{6.5}$$

The WCRT of the crossing job J_i of task τ_i is:

$$\begin{aligned} R_i(MS) = C_i(L_i) &+ \sum_{j \in hpH(i)} \left\lceil \frac{R_i(MS)}{T_j} \right\rceil \cdot C_j(HI) \\ &+ \sum_{k \in hpL(i)} \left(n_k^{LO} + \left\lceil \frac{R_i(MS) - n_k^{LO} \cdot T_k(LO)}{T_k(HI)} \right\rceil\right) \cdot C_k(LO) \end{aligned} \tag{6.6}$$

where $n_k^{LO} = \left\lfloor \frac{R_i(LO)}{T_k(LO)} \right\rfloor$. Note that the last term in Equation (6.6) is never negative since $R_i(LO) \leq R_i(MS)$.

The taskset is schedulable by EMC if :

$$\begin{cases} \forall (i | L_i = LO), \;\; R_i(LO) \leq D_i \wedge \max(R_i(MS), R_i(HI)) \leq D_i(HI) \\ \forall (i | L_i = HI), \;\; \max(R_i(LO), R_i(HI), R_i(MS)) \leq D_i. \end{cases} \tag{6.7}$$

Consider the two special cases mentioned earlier: if the period scaling factor

$k_i = T_i(HI)/T_i(LO) = 1$, then Equation (6.5) becomes:

$$\left\lceil \frac{R_i(MS)}{T_k(HI)} \right\rceil \cdot C_k(LO) = \left\lceil \frac{R_i(MS)}{T_k(LO)} \right\rceil \cdot C_k(LO) \tag{6.8}$$

and Equation (6.6) becomes:

$$R_i(MS) = C_i(L_i) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i(MS)}{T_j} \right\rceil \cdot C_j(HI) + \sum_{k \in hpL(i)} \left\lceil \frac{R_i(MS)}{T_k(LO)} \right\rceil \cdot C_k(LO) \tag{6.9}$$

This is the same as the classic Response-Time Analysis equation for a regular task model, with each task's WCET equal to its WCET at its own criticality level in the mixed-criticality model. Hence it verifies that the task model is equivalent to the non-mixed-criticality regular task model if $k_i = 1$.

If $k_i = \infty$, then Equation (6.5) becomes:

$$\left(\left\lfloor \frac{R_i(LO)}{T_k(LO)} \right\rfloor + 1 \right) \cdot C_k(LO) \tag{6.10}$$

and Equation (6.6) becomes:

$$R_i(MS) = C_i(L_i) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i(MS)}{T_j} \right\rceil \cdot C_j(HI) + \sum_{k \in hpL(i)} \left(\left\lfloor \frac{R_i(LO)}{T_k(LO)} \right\rfloor + 1 \right) \cdot C_k(LO) \tag{6.11}$$

Compare this to Equation (2.18) for AMC-rtb, we can see that Equation (6.6) is a slightly more pessimistic version of (2.18) (with $\lfloor\rfloor+1$ instead of $\lceil\rceil$ operator

in the last term). Hence it verifies that the task model is equivalent to the AMC model if $k_i = \infty$.

6.4 Priority Assignment with Audsley's Algorithm

Audsley's Optimal Priority Assignment (OPA) algorithm [105] is a well-known algorithm for priority assignment in fixed-priority scheduling. The algorithm tries to assign the lowest priority level (= 1) to a task. Once a task is feasibly assigned the lowest priority, remove it from consideration, and iterate until all tasks are assigned feasible priorities, and the algorithm succeeds; or no task can be feasibly assigned the lowest priority, and the algorithm fails. The OPA algorithm is applicable if the following four conditions are satisfied [105, 106]:

- A task's WCRT is dependent on the set of higher priority tasks, but not on the relative priority ordering of those tasks.
- A task's WCRT may be dependent on the set of lower priority tasks, but not on the relative priority ordering of those tasks.
- When priorities of any two tasks are swapped, the WCRT of the task being assigned a higher priority cannot increase with respect to its previous value.
- When priorities of any two tasks are swapped, the WCRT of the task being assigned a lower priority cannot decrease with respect to its previous value.

An inspection of the scheduling equations for both of both AMC and EMC indicates that these properties hold true, hence OPA is applicable to both AMC and EMC.

6.5 Optimization Algorithms

Recall that our optimization objective is to minimize the total system cost, defined as the sum of costs of all edges with a unit delay, while guaranteeing schedulability. We present two optimization algorithms for assigning task priorities, and selectively adding delays to edges to generate a schedulable semantics-preserving implementation: an optimal BnB algorithm that guarantees to find the solution with minimal cost and an efficient greedy heuristic algorithm that can find close-to-optimal solutions.

6.5.1 The Branch-and-Bound algorithm

We note two special cases: the AllDelay configuration with delays added to all edges ($\mathcal{E}_d = \mathcal{E}$), where all intertask precedence constraints are eliminated; and the NoDelay configuration with no delays added to any edge ($\mathcal{E}_d = \phi$), where all intertask precedence constraints are preserved. All other delay configurations lie between the two special cases. For a given SR model, the AllDelay configuration has the highest cost and the best schedulability, and the NoDelay configuration has the lowest cost and the worst schedulability. If the NoDelay configuration is schedulable, then it is the optimal solution; otherwise, the BnB algorithm starts from the root node of the NoDelay configuration and gradually adds edge delays to find the optimal delay configuration

that is barely schedulable and minimizes the total system cost. Each node in the search tree represents a unique delay configuration. We consider two alternative approaches to constructing the BnB search tree:

- Consider each edge and make a 0-1 decision, with 0 indicating no delay, and 1 indicating one delay on that edge. Each node of the resulting search tree has two children nodes, corresponding to either 0 (no delay) or 1 (one delay). Each child node has the same delay configuration as its parent node plus one additional delay on a new edge. In order to minimize the total system cost, the edges in the SR model are considered in the order of increasing cost, reflecting the preference to add delay to an edge with lower cost than another edge with higher cost. The full search tree, if no branches are pruned, is a balanced binary tree.

- Consider each delay as a token, and make a decision on which edge to place it on. Each node of the resulting search tree may have multiple children nodes, corresponding to the different choices of the next candidate edge to place a delay token onto. Again, each child node has the same delay configuration as its parent node plus one additional delay on a new edge. For example, The root node is the NoDelay configuration. It has E child nodes, where E is the total number of edges in the SR model, reflecting the E possible choice edges to place a delay token onto. The full search tree, if no branches are pruned, is a skewed tree that is deep and bushy on the left-hand side and shallow and sparse on the right-hand side.

One important factor that determines the efficiency of BnB search is the bounding function for effective search tree pruning, which determines the lower bound for each search tree node, as the lowest cost of all possible configurations that can be obtained by expanding this node and exploring future configurations starting from this node's configuration. The search tree is pruned at this node if its lower bound exceeds the cost of the current best configuration seen so far. While the first approach is simple and intuitive, it is difficult to design a good bounding function. We adopt the second approach, as proposed in [84].

To determine the ordering among edges for adding delays to, we define Λ_e as the list of all edges in the SR model sorted in the order of increasing cost; in case of a tie, where multiple edges have the same cost, the same-cost edges are sorted in decreasing execution rate of their sender tasks, reflecting the preference to add delay to an edge whose sender task has a higher execution rate (smaller period), since each added delay implies increased end-to-end latency on the edge by an amount equal to the sender task period. Each search tree node p corresponds to a delay configuration, that is, the set of edges with one delay on each edge, denoted as p.config. Each child node c of the parent node p has the configuration obtained by adding one additional delay to an unexplored edge in parent node p's configuration, where the list of unexplored edges consists of all edges in the sorted list Λ_e after (with costs higher than or equal to) all edges with delays in node p's configuration. Node c's cost is computed as the sum of costs of all the edges with delays in its configuration; its bound is computed as node p's cost plus cost of the next edge, also the minimum-cost edge, in the list of unexplored edges. This prevents adding redundant nodes to

the search tree with configurations that have already been considered before. The BnB search proceeds according to A* search, that is, at any time during the search, we always expand the node on the frontier of the search tree (the set of candidate nodes to be expanded next) with the lowest bound. In order to implement Best First Search (BFS), we keep the list of candidate nodes in a priority queue OPEN sorted by increasing bound. Algorithm 6.1 shows the pseudocode for the BnB algorithm:

Algorithm 6.1: Branch-and-Bound Algorithm BnB()

```
1: Λ_e.sortByIncreasingCost()
2: node_0.delays=φ
3: OPEN.enqueue(node_0)
4: optimalsol.cost ← ∞
5: while OPEN not empty do
6:    p = OPEN.dequeue()
7:    if p.bound < optimalsol.cost then
8:       for each child c of p do
9:          if OPA4SR(c.Config) then then
10:            if c.cost < optimalsol.cost then
11:               optimalsol = c
12:               Remove from OPEN all candidate configurations with bounds
                  higher than optimalsol.cost
13:            end if
14:         else
15:            if c.bound < optimalsol.cost then
16:               OPEN.enqueue(c)
17:            end if
18:         end if
19:      end for
20:   end if
21: end while
```

- Lines 1-4: Initialize the sorted list of edges Λ_e, the root node $node_0$, the priority queue OPEN sorted by increasing bound, and the lowest cost of the configurations seen so far (optimalsol.cost).

- Lines 5-21: The main loop of the algorithm, which terminates when OPEN is an empty set, that is, all search tree nodes have either been expanded or pruned.

- Line 6: Take the configuration in front of the priority queue OPEN (p = OPEN.dequeue()).

- Lines 7-20: If p's bound is less than the minimum cost among all configurations seen so far (p.bound < optimalsol.cost), the node is expanded further to search for a possible better configuration with lower cost than optimalsol.cost.

- Lines 8-19: Consider each child node c of the parent node p.

- Lines 9-13: If the procedure OPA4SR() finds a feasible priority assignment for c's configuration, and if c's cost is lower than optimalsol.cost, then set c to be the new optimalsol. In addition, remove from OPEN all candidate configurations with bounds higher than optimalsol.cost, since they cannot produce any better configurations.

- Lines 14-18: If the procedure OPA4SR() fails to find a feasible priority assignment for c's configuration, and if c's bound is lower than optimalsol.cost, then it may be possible to find a better configuration by expanding its child nodes, so add c to OPEN to be expanded next.

Note that line 12 (removing from OPEN all candidate configurations with bounds higher than optimalsol.cost), and line 15 (checking of c.bound < optimalsol.cost) are both redundant given the check on line 7 for c.bound < op-

timalsol.cost. Line 12 is for early removal of unpromising nodes fromOPEN, and line 15 is for preventing unpromising nodes from being added into OPEN. They are both performance optimizations that can be omitted without affecting algorithm correctness.

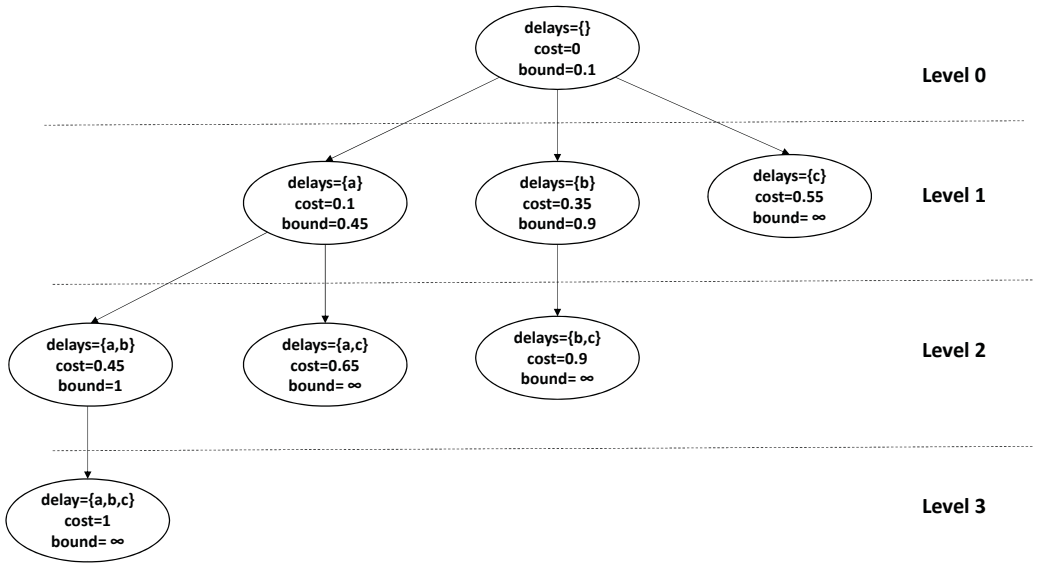


Fig. 6.4: Example of a Branch-and-Bound search tree.

As an example, Figure 6.4 shows an example of the BnB search tree for a system with 3 edges a, b and c, with costs of 0.1, 0.35, and 0.55, respectively. They are sorted by increasing cost, with Λ_e as the sorted list of edges a, b, c. The search starts from the root node of the *NoDelay* configuration, and first tries to add a delay to edge a, generating the configuration with Delays={a}, Cost=0.1. The lower bound of all possible solutions from the current search tree node is equal to cost of this node plus the minimum edge cost in the remaining edges in the list Λ_e, which is equal to 0.1+min(0.35,0.55)=0.45. The other search nodes can be constructed similarly. Note that the rightmost

search tree node has delays={c} and cost=0.55. There is no edge with cost higher than c in OPEN, hence it should not have any children node. We assign its bound to be a large value (represented by infinity ∞ in the figure) higher than the maximum solution cost, to indicate to the algorithm that it should stop and not explore any children of this node.

Now we turn to the priority assignment algorithm for SR models. Algorithm 6.2 shows the algorithm OPA4SR(), that is, the Optimal Priority Assignment algorithm for SR models. It takes as argument a given delay configuration $\mathcal{E}_d$, and tries to assign priority level k to a task so that the task is feasible, starting from the lowest priority-level 1 to the highest priority-level $|\mathcal{N}|$, assuming each task $\tau_i \in \mathcal{N}$ is assigned a unique priority. Different from the original OPA algorithm, OPA4SR() must take into account the priority assignment constraints imposed by feedthrough edges, that is, the sender must have higher priority than the receiver if there is a feedthrough edge between them. For the AllDelay configuration of an SR model, where all intertask dependency relationships are eliminated, the OPA can be applied directly, where all remaining tasks whose priorities have not been assigned are possible candidates to be assigned the lowest available priority. For all other configurations of an SR model, a task N_i whose priority has not been assigned can be a candidate to be assigned the next lowest available priority, only if there is no feedthrough edges $N_i \rightarrow N_j$ to a receiver task N_j that has not been assigned a priority yet. In other words, the order of priority assignment is from receiver tasks to sender tasks in reverse order of the feedthrough edges. We define the set of candidate tasks $\mathcal{N}_c$ to be assigned the next lowest available priority, as

Algorithm 6.2: OPA4SR($\mathcal{E}_d$)

```
1: for k = 1 to |𝒩| do
2:    Construct the set of candidate tasks 𝒩_c ⊆ 𝒩 based on the current delay
      configuration.
3:    for each τ_i ∈ 𝒩_c do
4:       if τ_i is schedulable at priority level k then
5:          p_i = k
6:          𝒩_c.remove(τ_i)
7:          break
8:       end if
9:    end for
10:   if no task is schedulable at priority level k then
11:      return FALSE
12:   end if
13: end for
14: return TRUE
```

the set of tasks with no feedthrough edges $N_i \rightarrow N_j$ to receiver tasks that have not been assigned priorities yet. If any task $\tau_i \in \mathcal{N}_c$ can be feasibly assigned the next lowest priority k, then assign $p_i = k$, remove τ_i from $\mathcal{N}_c$, and try the next higher-level priority k+1 at the next iteration; if none of the tasks in $\mathcal{N}_c$ can be feasibly assigned the next lowest priority k, then declare the task set to be infeasible with any priority assignment. The main difference from the original OPA lies in lines 2 and 3, where the pool of candidate tasks $\mathcal{N}_c$ needs to be updated at every iteration of the for loop, that is, after an additional task is feasibly assigned a priority. This difference affects the order in which task priorities are assigned, not the schedulability analysis equations for AMC and EMC, or the four conditions for OPA to be applicable. Therefore, OPA4SR() is applicable to priority assignment for a mixed-criticality SR model with either an AMC or EMC scheduling algorithm.

Consider a SR model with N nodes and E low-rate-to-high-rate dependency edges. The optimal algorithm needs to explore at most 2^E different delay configurations. For each configuration, the function `Audsley_Schedulable()` is called, which performs a maximum of $N(N+1)/2$ schedulability tests. So the overall worst-case running time of the algorithm is $O(2^E \times N^2)$ number of schedulability tests. (The schedulability tests in Section 2.2 based on recursive equation solving is pseudo-polynomial, but it is generally very efficient in our experience). Due to its worst-case exponential running time, BnB is not scalable to large systems, hence we present an efficient heuristic algorithm as an alternative to BnB.

6.5.2 The heuristic algorithm

Algorithm 6.3 is an efficient greedy algorithm for simultaneous priority assignment and delay optimization. We define each task's cost as the sum of the costs of its outgoing feedthrough edges to receiver tasks that have not been assigned priorities yet. $\mathcal{N}_u$ is the list of tasks that have not been assigned priorities yet, sorted by increasing cost. In case of a tie, where multiple tasks have the same cost, the same-cost tasks are sorted in increasing execution rate (decreasing period), reflecting the preference to assign lower priority to tasks with lower execution rate (larger period), consistent with rate-monotonic policy (even though rate-monotonic policy is no longer optimal in this context, it is still a useful heuristic). $\mathcal{N}_u$ is initialized to be the full task set $\mathcal{N}$, and should be gradually reduced to the null list if a feasible priority assignment is found. $\mathcal{E}_u$ is the set of candidate edges for delay addition, initialized to

Algorithm 6.3: *Heuristic*()

```
1: {Phase 1}
2: N_u ← N, E_u ← E, sysCost ← 0
3: for k = 1 to |N| do
4:    compute costs of tasks in (N_u) and sort N_u by increasing cost
5:    for each τ_i ∈ N_u do
6:       if τ_i is schedulable at priority level k then
7:          p_i = k
8:          N_u.remove(τ_i)
9:          E_u.add_delay(τ_i.outgoingFTEdges)
10:         E_u.remove(τ_i.incomingFTEdges)
11:         sysCost += τ_i.cost
12:         break
13:      end if
14:   end for
15:   if no task τ_i ∈ N_u is schedulable at priority level k then
16:      return FAILURE
17:   end if
18: end for
19: E_d ← E_u
20: {Phase 2}
21: Sort all edges with delays E_d by decreasing cost
22: for each edge E_i ∈ E_d do
23:   E_d.remove(E_i)
24:   if OPA4SR(E_d)=FALSE then
25:      E_d.add(E_i)
26:   else
27:      sysCost -= E_i.cost
28:   end if
29: end for
30: return SUCCESS
```

be the full set of edges $\mathcal{E}$, and should be gradually reduced to $\mathcal{E}_d$, the set of edges with delays on them, each pointing from a lower-priority sender task to a higher-priority receiver task. (The subscript u means unassigned.)

Lines 1-19 describe Phase 1, an OPA-based algorithm that performs priority assignment from the lowest to the highest level by assigning the next lowest

priority level to the feasible task with minimum cost. Lines 20-30 describe Phase 2, which attempts to remove some edge delays while still guaranteeing schedulability. More specifically, Algorithm 6.3 proceeds as follows:

1. Line 2: Initialize $\mathcal{N}_u$, $\mathcal{E}_u$, and the system cost.

2. Line 4: Compute costs of tasks in $\mathcal{N}_u$, and sort them by increasing cost.

3. Lines 5-14: If any task $\tau_i \in \mathcal{N}_u$ can be feasibly assigned the next lowest priority k, then assign $p_i = k$; remove τ_i from $\mathcal{N}_u$. At this point, all of τ_i's neighbor tasks connected by feedthrough edges have not been assigned priorities, and τ_i has lower priority than all of its neighbor tasks. τ_i's outgoing feedthrough edges connect lower-priority sender task τ_i to higher-priority receiver tasks, so add delays to them, and add costs of these edges, which sum to τ_i.cost, to the system cost; τ_i's incoming feedthrough edges connect higher-priority sender tasks to lower-priority receiver task τ_i , so remove all such edges from $\mathcal{E}_u$, since no delays should be added to such edges.

4. Lines 15-17: If none of the tasks in $\mathcal{N}_u$ can be feasibly assigned the next lowest priority k, then declare the task set to be unschedulable and return FAILURE.

5. Line 19: Set $\mathcal{E}_d$ to be equal to $\mathcal{E}_u$.

6. Lines 20-30: Attempt to remove delays from the set of edges with delays $\mathcal{E}_d$. For each edge $E_i \in \mathcal{E}_d$ in the order of decreasing cost (in case of a tie, the same-cost edges are sorted in decreasing execution rate of their

sender tasks), remove its edge delay and invoke OPA4SR() in Algorithm 6.2 for priority assignment. If OPA4SR() fails to find a feasible priority assignment, then add back the edge delay; otherwise, actually remove the delay and subtract the edge cost from sysCost. Continue to try to remove delay from the next edge, until all edges in $\mathcal{E}_d$ have been processed.

An illustrative example

To illustrate the heuristic algorithm, consider the example in Figure 6.5, with timing parameters in Table 6.3. The output block τ_3 is HI-criticality, and the output block τ_4 is LO-criticality; the non-output blocks τ_1 and τ_2 are both assigned HI criticality, since they are predecessors of HI-criticality output block τ_3.

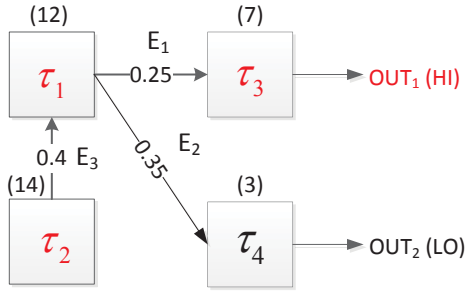


Fig. 6.5: An example mixed-criticality synchronous program. The number in parentheses above a block denotes its period.

Table 6.3: Timing parameters for the example in Figure 6.5.

τ_i	L_i	T_i	D_i	$C_i(LO)$	$C_i(HI)$
τ_1	HI	12	12	2	3
τ_2	HI	14	14	1	2
τ_3	HI	7	7	1	2
τ_4	LO	3	2	1	-

Phase 1: Try to find a feasible priority assignment:

1. Try to find the task that can be assigned the lowest priority. We compute the cost of each task as the sum of the cost of all its outgoing edges, and we can get $cost(\tau_1) = 0.6$, $cost(\tau_2) = 0.4$, $cost(\tau_3) = 0$,$cost(\tau_4) = 0$. Then we search for the task that can be assigned the lowest priority in the order of increasing cost, which is $\tau_3, \tau_4, \tau_2, \tau_1$. WCRT of τ_3 if assigned the lowest priority is $R_3^* = 9 > D_3 = 7$; WCRT of τ_4 if assigned the lowest priority is $R_4^{LO} = 5 > D_4 = 2$; WCRT of τ_2 if assigned the lowest priority is $R_2^* = 11 \leq D_2 = 14$, we assign τ_2 the lowest priority. Because there is a writer-to-read dependency between τ_2 and τ_1 and writer τ_2 has lower priority than reader τ_1, an edge delay is added between them, with a cost of 0.4.

2. Try to find the lowest priority task among the remaining tasks, in the order of increasing cost τ_3, τ_4, τ_1. WCRT of τ_3 if assigned the lowest priority is $R_3^* = 7 \leq D_3 = 7$, we assign τ_3 the lowest priority. Because writer τ_1 has higher priority than reader τ_3, an edge delay is not needed between them. Now we have $cost(\tau_1) = 0.35, cost(\tau_4) = 0$.

3. Try to find the lowest priority task among the remaining tasks, in the order of τ_4, τ_1. WCRT of τ_4 if assigned the lowest priority $R_4^{LO} = 3 > D_4 = 2$; WCRT of τ_1 if assigned the lowest priority is $R_1^* = 3 \leq D_1 = 12$, we assign τ_1 the lowest priority. Because there is a writer-to-read dependency between τ_1 and τ_4 and writer τ_1 has lower priority than reader τ_4, an edge delay is added between them, with a cost of 0.35.

4. WCRT of τ_4, with the highest priority, is $R_4^{LO} = 1 \leq D_4 = 2$. So we have found a feasible priority assignment, from lowest to highest priority $\tau_2, \tau_3, \tau_1, \tau_4$, with total system cost $0.4 + 0.35 = 0.75$.

Phase 2: Try to remove some unnecessary delays. There are two delays, one on the edge E_{21} from τ_2 to τ_1 with cost of 0.4; the other on the edge E_{14} from τ_1 to τ_4, with cost of 0.35. We try to remove the delays in descending order of cost.

1. Try to remove the delay on the edge E_{21} from $\mathcal{E}_d = \{E_{21}, E_{14}\}$. If the delay on the edge E_{21} is removed, OPA4SR($\mathcal{E}_{14}$) as shown in Algorithm 6.2 fails, so we cannot remove the delay on the edge E_{21}.

2. Try to remove the delay on the edge E_{14} using OPA4SR($\mathcal{E}_{21}$). This attempt also fails.

Therefore, Phase 2 was not able to remove any edge delay from $\mathcal{E}_d = \{E_{21}, E_{14}\}$, and so the final result by the heuristic algorithm is a feasible priority assignment from lowest to highest priority $\tau_2, \tau_3, \tau_1, \tau_4$, with total delay cost 0.75.

Consider a SR model with N nodes and E low-rate-to-high-rate dependency edges. The first phase of the heuristic algorithm is based on Audsley's OPA and performs a maximum of $N(N+1)/2$ schedulability tests. In the second phase, the heuristic algorithm needs to process at most E edges. For each edge, the function `Audsley_Schedulable()` is called, which performs a maximum of $N(N+1)/2$ schedulability tests. So the overall running time of the heuristic algorithm is $O(E \times N^2)$ number of schedulability tests, much more efficient than the exponential complexity of $O(2^E \times N^2)$ of the optimal algorithm.

6.6 Experimental Results

We use Task Graph For Free (TGFF) [107] to generate random DAGs of SR models for performance evaluation. The maximum fan-in of each node is 3 and the maximum fan-out is 2. 1000 random systems are generated for each data point. We assume that each node corresponds to a task, so each SR model corresponds to a taskset with size equal to the number of nodes in the model.

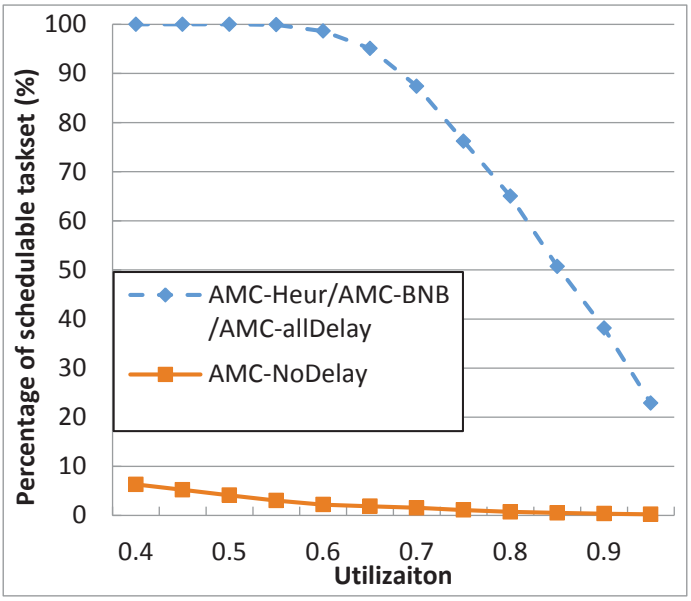


Fig. 6.6: Schedulability of different algorithms vs. U_{LO} for AMC.

Taskset generation is controlled by the following parameters: number of tasks in each taskset N (default=12 if unspecified); system CPU utilization in LO-criticality mode U_{LO}; percentage of HI-criticality tasks P_{HI} (default=50% if unspecified); Each HI-criticality task τ_i's *criticality factor*, defined as $F_i = C_i(HI)/C_i(LO)$ (default=2 if unspecified); Each LO-criticality task τ_i's period scaling factor $k_i = T_i(HI)/T_i(LO)$ (default=2 if unspecified). In general, F_i

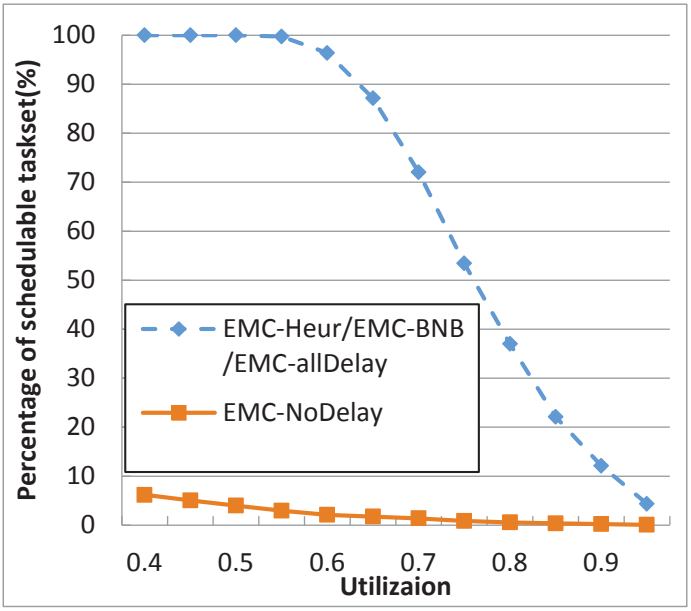


Fig. 6.7: Schedulability of different algorithms vs. U_{LO} for EMC.

and k_i can have different values for different tasks, but we assume them to be the same for all tasks in the experiments, denoted as F and k. Each task τ_i is generated as follows: its criticality level $L_i = HI$ with probability P_{HI}; its nominal period T_i is randomly selected from the set {10, 20, 40, 50, 100, 200, 400, 500, 1000} ms with uniform distribution; its LO-criticality utilization $(U_i(LO) = C_i(LO)/T_i)$ is generated with UUnifast [101], with maximum value of 49%; its LO-criticality WCET $C_i(LO) = U_i(LO) \cdot T_i$; its deadline $D_i = T_i$; its HI-criticality WCET $C_i(HI)$ is $\min(D_i, F_{HI} \cdot C_i(LO))$ if $L_i = HI$; $C_i(HI) = C_i(LO)$ if $L_i = LO$. Recall that unit delays are always added on all high-rate-to-low-rate dependency edges, so we only consider delays added to the low-rate-to-high-rate dependency edges. Each low-rate-to-high-rate dependency edge is assigned a random cost according to the uniform distribution such that the sum of the costs equals 1, to simulate the relative impact of functional delays

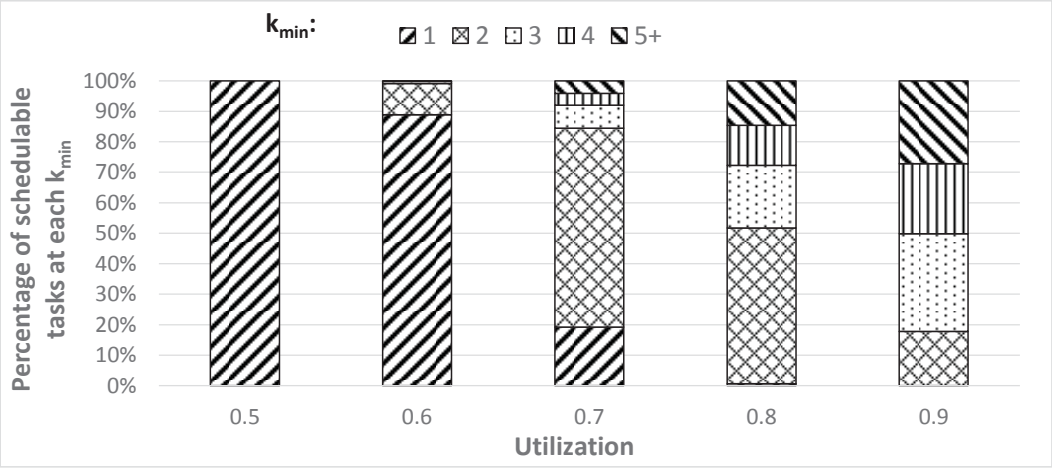


Fig. 6.8: Schedulability of EMC vs. minimum period scaling factor $k_{\min}$, for different U_{LO}.

on different edges on application-level control performance. The optimization objective is to minimize the total *system cost*, defined as sum of all costs of low-rate-to-high-rate dependency edges with a unit delay. Two extreme cases are the *AllDelay* configuration, where all edges have unit delays on them, with total cost of 1; and the *NoDelay* configuration, where no low-rate-to-high-rate dependency edge has a unit delay. Obviously, for a given SR model, the *AllDelay* configuration has the highest cost and the best schedulability, and the *NoDelay* configuration has the lowest cost and the worst schedulability. If the *NoDelay* configuration is schedulable, then it is the optimal solution; otherwise, our objective is to find the delay configuration that is "barely schedulable" and has the minimum cost. The experiments are performed on a computer with CPU speed of 2.8 GHz, and 8 GB RAM.

We first evaluate how added delays can improve system schedulability. We compare the following approaches: the optimal branch-and-bound based al-

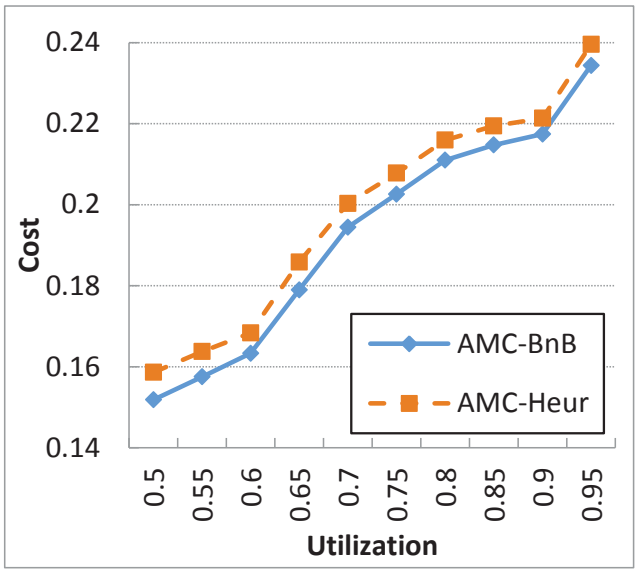


Fig. 6.9: System cost vs. U_{LO} for AMC.

gorithm (AMC-BNB/EMC-BNB), the heuristic algorithm (AMC-Heur/EMC-Heur), the configurations (AMC-AllDelay/EMC-AllDelay), and the configurations (AMC-NoDelay/EMC-NoDelay). The number of tasks in each taskset is 12. Figures 6.6 and 6.7 plot the percentage of schedulable tasksets vs. the system CPU utilization in LO-criticality mode U_{LO}. We can see that the percentage of schedulable tasksets is much higher for the implementations with delays than those without delays. Among the approaches based on AMC, it is obvious that AMC-AllDelay, where all dependency relationships are eliminated, has the best performance in terms of schedulability; and AMC-NoDelay has the worst preformance. AMC-BNB and AMC-Heur have the same performance as AMC-AllDelay, since AMC-BNB and AMC-Heur try to remove delays while guaranteeing schedulability, so in the worst-case, no delays can be removed, and they yield the same solution as AMC-AllDelay. Similarly, EMC-BNB, EMC-Heur and EMC-AllDelay have the same performance in terms

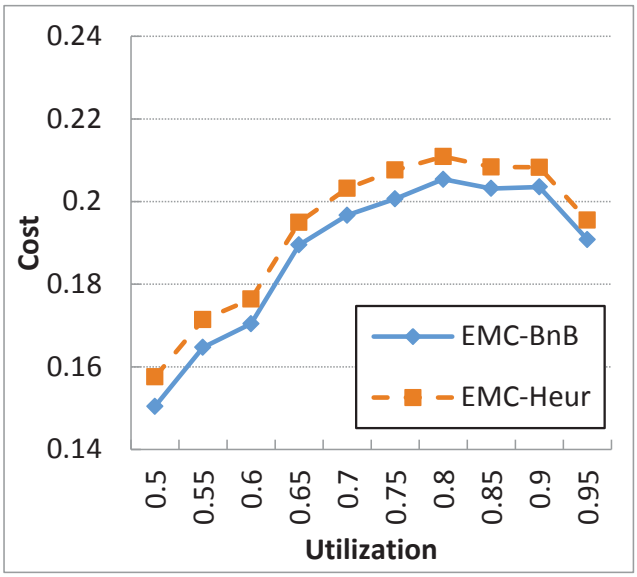


Fig. 6.10: System cost vs. U_{LO} for EMC.

of schedulability. The AMC-based algorithms generally have better schedulability than the EMC-based algorithms for the same taskset, since all LO-criticality tasks are dropped in HI-criticality mode in AMC, resulting in lower CPU utilizations in HI-criticality mode than EMC.

We then evaluate the impact of the EMC model on schedulability. We only consider tasksets that are *schedulable with AMC-AllDelay*, by discarding all tasksets that are not schedulable with AMC-AllDelay during taskset generation. We define the *minimum period scaling factor* $k_{\min}$ as the minimum k such that the system is schedulable when all LO-criticality tasks have $T_i(HI) = k \cdot T_i(LO)$. Larger value of $k_{\min}$ indicates more QoS degradation for LO-criticality tasks in HI-criticality mode. The number of tasks in each taskset is 20. Figure 6.8 shows the percentage of schedulable tasksets at each $k_{\min}$ for different U_{LO}. As the system utilization becomes higher, more systems require

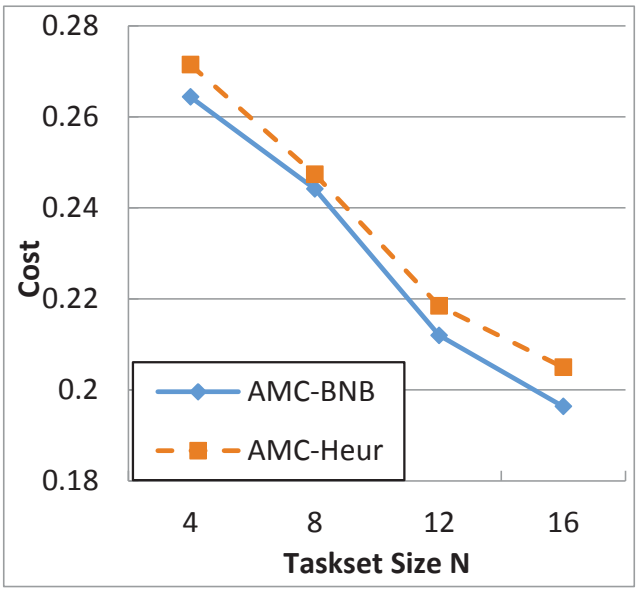


Fig. 6.11: System cost vs. taskset size for AMC.

a larger $k_{\min}$ to be schedulable, i.e., more QoS degradation of LO-criticality tasks is needed to make the system schedulable.

In the following experiments, we only consider tasksets that are *schedulable with both AMC-AllDelay and EMC-AllDelay*, by discarding all tasksets that are not schedulable with AMC-AllDelay during taskset generation.

Figures 6.9 and 6.10 plot system cost vs. LO-criticality CPU utilization U_{LO}. For every taskset where the optimal algorithm finds a feasible solution, the heuristic algorithm is also able to find a feasible solution, and the solution it finds (dashed line) has system cost close to the optimal solution (solid line) (about 3% higher). The system cost increases with U_{LO} for AMC, since it is necessary to add more delays to make the system schedulable with higher CPU utilization; the system cost increases with U_{LO} for EMC, but shows a small decrease when U_{LO} goes from 0.9 to 0.95. We find this decrease to be due

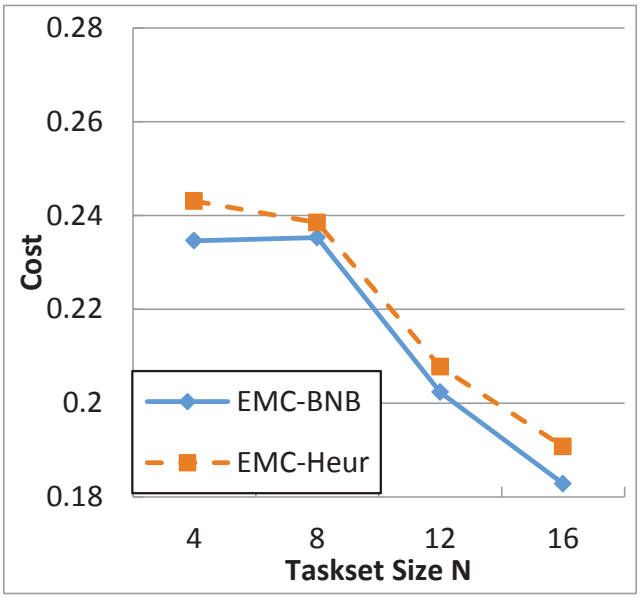


Fig. 6.12: System cost vs. taskset size for EMC.

to the biased generation of tasksets: since we only consider tasksets that are schedulable with both AMC and EMC, most randomly-generated tasksets with $U_{LO} = 0.95$ are discarded due to non-schedulability. The surviving tasksets tend to have relatively few low-rate-to-high-rate dependency edges, so it is not necessary to add many edge delays to make them schedulable, hence the system cost is not large.

For Figures 6.11, and 6.12, the system cost decreases with increasing taskset size N. Increasing taskset size N cause the following: the number of edges become larger, the cost of each edge becomes smaller, and the average value of each task's CPU utilization ($U_i(LO)$ becomes lower. Hence the optimization algorithm has more freedom in choosing the edges to place the delays. Combined with the effect due to biased generation of tasksets mentioned before, the overall effect is that the system cost decreases with increasing N.

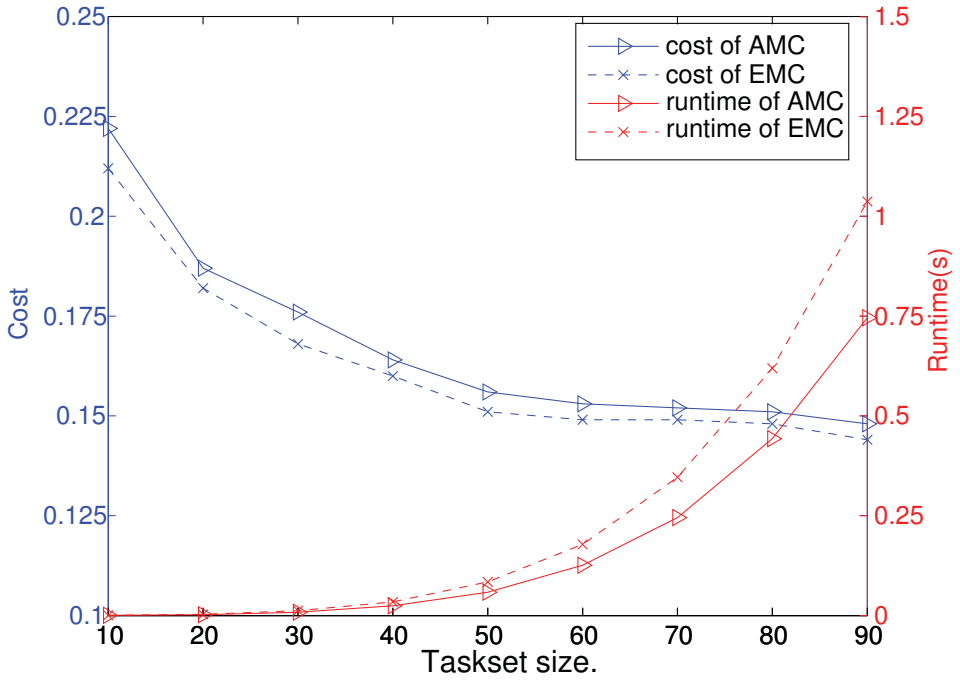


Fig. 6.13: System cost and runtime of the heuristic for larger systems.

In Figure 6.13, we vary the task set size N from 10 to 90, keeping $U_{LO} = 0.8$, and plot the total system cost, algorithm running time of the heuristic algorithm. (The optimal algorithm cannot scale to large task sets, so its solutions are not plotted here.) The results verify the efficiency of the heuristic algorithm, which takes only around 1 second for taskset size of 90.

6.7 Conclusions

In this chapter, we consider the optimization of the multitask implementation for mixed-criticality synchronous models with fixed-priority scheduling on a uniprocessor. To provide minimum service level for low criticality tasks, we define the Elastic Task Model, and present schedulability analysis for it. We

develop an exact algorithm based on Audsley's Optimal Priority Assignment algorithm for task priority assignment, and use branch-and-bound search for assigning delays, with the objective of minimizing the weighted sum of the functional delays. We also propose an efficient heuristic. Experiments at different parameter settings showed that the proposed heuristic obtains close-to-optimal results, performing only 3% worse than the optimal branch-and-bound algorithm. Our experiments also showed that the heuristic is scalable to large systems as it takes only around 1 second for systems of 90 tasks.

Chapter 7

Conclusion and Future Work

Embedded systems will continue to shape our lives in ways that a few decades ago were not possible. A large part of these systems are real-time systems that have externally defined timing constraints. Advances in embedded design open up new application opportunities by enabling systems that process more information faster, cost less, and tolerate more faults. This can be done in two different ways: improving the platform or improving the design process. In this thesis, we use the latter approach; assuming a fixed platform, propose improved models, analysis, and design techniques to produce more efficient and more robust systems. We summarize our key contributions next:

7.1 Key Contributions

7.1.1 Resource-sharing multicores

With the prominence of multicore architectures in embedded computing today, the first additional challenge introduced by these systems is allocating

tasks across the different cores efficiently. This is complicated by resource sharing as tasks in multicore systems often share resources such as variables and data structures. Access to the shared resource can be managed by different mechanisms with varying blocking and memory penalties. Previous work has addressed the problems of task allocation and resource management separately leading to inefficient solutions.

In Chapter 3, we proposed to jointly optimize task allocation and the selection of resource protection mechanisms. Two methods were presented to solve this optimization problem; an optimal Mixed Integer Linear Programming (MILP) and a sub-optimal but efficient heuristic. Experiments showed that the proposed approach significantly extends the range of systems that can be scheduled on a given platform. More specifically, the critical utilization (maximum utilization at which at least 95% of systems are found schedulable) is improved from the 57%-70% range in state-of-the-art task allocation heuristics to the 76%-88% range.

7.1.2 Multicore mixed-criticality systems

In Chapters 4, 5 and 6, we discussed the design of Mixed-Criticality Systems (MCS) that have been proposed to address the need to certify embedded designs. MCS design balances the requirements of the certification authorities to provide sufficient computing resources to handle worst case scenarios with the needs of manufacturers for efficiency and cost reduction. In Chapter 4, we showed that when MCS are implemented on a multicore architecture, the variations in execution times at different criticality levels can lead

to inefficient designs. To address this issue, we proposed the Dual-Partitioned Mixed-Criticality (DPM) task allocation approach. The DPM approach allows the system to have different partitions at different modes, while maintaining the property that in any stable mode, the system is fully partitioned. DPM provides more efficiency for non-critical tasks while maintaining safety requirements for critical ones. Experiments showed a 17% improvement in systems schedulability using DPM for systems of 80% utilization or more.

7.1.3 Fault-tolerant mixed-criticality systems

In Chapter 5, we argued that reliability must be pro-actively considered in the design process of MCS since they contain a safety critical part. To this end, we proposed a new model and analysis for MCS that simultaneously address hardware transient faults, certification and quality of service. To aid in the use of the new model in developing systems, we proposed a Design Space Exploration (DSE) approach that is based on the proposed model and supports a variety of fault-tolerance mechanisms. For tolerating permanent hardware faults, we extended the standard MCS model and analysis to support the failure of one processor in the system. An accompanying MILP-based DSE approach is also proposed that is shown to achieve a a 3.2X improvement in schedulability.

7.1.4 Implementing synchronous reactive models of mixed-criticality systems

In Chapter 6, we focused on model-based design, specifically, using the popular Synchronous Reactive (SR) model and proposed algorithms to optimize the implementation of MCS SR models onto the underlying platform. Our objective was to find the schedulable implementations that preserve the semantics of the SR model (by adding functional delays when necessary to preserve communication flows) while reducing the performance penalty of the added delays. To this end, we proposed an optimal Branch-and-Bound based algorithm and an efficient heuristic.

7.2 Summary of Best Practices

Throughout this work, we made a few observations that could be useful for future industrial or academic works. We summarize them below:

- The efficiency of multicore architectures can be significantly improved by using different resource protection mechanisms and having knowledge of the mechanisms used for each resource when making task allocation decisions.

- Partitioned scheduling can be inefficient in mixed-criticality systems due to their inherent nature (having different utilizations in different modes). Semi-partitioned scheduling can improve efficiency while keeping the desired features of partitioned scheduling.

- To achieve both efficiency and reliability, it is important to think about reliability early in the design process. Scheduling Models, analysis techniques, and design processes explicitly modelling fault-tolerance can help in this regard.

- On-demand redundancy can have significant advantages in a mixed-criticality context. Achieving efficiency in MCS requires flexibility which can be provided by ODR.

7.3 Future Work

The research presented in this thesis can be further developed to increase its applicability to embedded designs. Possible future research directions include:

7.3.1 Resource-sharing multicore mixed-criticality systems

In Chapter 3, we studied task allocation in resource sharing multicore systems. Resource locking was managed using the Multiprocessor Priority Ceiling Protocol (MPCP) and the Multiprocessor Stack resource Policy (MSRP). For resource-sharing MCS, a different approach based on resource servers is proposed in [44]. While this approach has strong guarantees for isolation, it can be costly. It is worthwhile to investigate whether MPCP or MSRP is applicable in a mixed-criticality context and whether changes are needed to these protocols to suit an MCS context. An important issue would be providing isolation with MPCP/MSRP. Furthermore, the issue of resource-aware task allocation will need to be re-investigated.

7.3.2 Platform variations

In this thesis, we focused on single core and homogeneous multicore platforms. Experiments were often conducted on systems composed of 1-8 cores. It would be interesting to investigate how the proposed algorithms would scale to 16-32 cores, or to many core platforms. Another interesting dimension to these problems is the consideration of heterogeneous platforms where different cores have different capabilities and/or different speeds.

7.3.3 Considering multiple viewpoints

In industrial standards such as AUTOSAR, and ARINC 653, schedulability needs to be considered together with multiple viewpoints. Each viewpoint comes with own sets of constraints. It would be interesting to investigate how models and design approaches proposed in this thesis can be adapted to consider multiple viewpoints.

7.3.4 Considering and quantifying migration overheads

A common simplification made in many real-time studies is neglecting real system overheads such as migration overheads. A migration occurs when a task is moved at runtime from one core to another often requiring some processing time. Future work can focus on methods for quantifying migration overheads in real-time systems. The DPM approach in Chapter 4 and the permanent fault tolerant MCS design approach (Sections 5.8 - 5.12) can then make use of the calculated overheads. These works can be extended to model a non-negligible impact of migrations on the model and the design approaches.

7.3.5 MCS under multiple processor failure

In Chapter 5, we proposed a design approach for MCS that can tolerate the failure of one processor. Designers might need to design systems that tolerate the failure of multiple processors to satisfy more stringent reliability requirements. The analysis and DSE approach could be extended to cover the failure of any number of processors. Moreover, a scalable but efficient heuristic can be developed for DSE in large systems where an MILP solution might not be feasible.

7.3.6 A holistic fault-tolerant MCS model

A robust MCS design should be tolerant to any type of faults. It would be worthwhile to investigate whether a holistic model that combines both transient and permanent faults in addition to certification would be relevant or whether it would be too complex for practical use.

References

[1] Advanced Research, Technology for EMbedded Intelligence, and Systems. Embedded / Cyber-Physical Systems ARTEMIS Major Challenges: 2014-2020. December 2013.

[2] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[3] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[4] `http://http://www.nhtsa.gov`.

[5] `http://www.forbes.com/sites/jimgorzelany/2014/03/26/automakers-with-the-lowest-and-highest-recall-rates`, . Accessed: 2016-11-21.

[6] `http://www.marketwatch.com/story/record-number-of-cars-recalled-in-2015-2016-01-21`, . Accessed: 2016-11-21.

[7] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[8] `https://auto.luxoft.com/uth/timing-analysis-tools/`.

[9] `https://www.absint.com/ait/index.htm`.

[10] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4): 35, 2011.

[11] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243. IEEE, 2007.

[12] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep., Seventh edition, January 2016*. URL `https://www-users.cs.york.ac.uk/burns/review.pdf`. Accesses December 2016.

[13] Sudarshan K Dhall and CL Liu. On a real-time scheduling problem. *Operations research*, 26(1):127–140, 1978.

[14] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 469–478. IEEE, 2009.

[15] Marco Di Natale, Liangpeng Guo, Haibo Zeng, and Alberto L. Sangiovanni-Vincentelli. Synthesis of Multi-task Implementations of Simulink Models with Minimum Delays. *IEEE Transactions on Industrial Informatics (TII)*, 6(4):637–651, 2010.

[16] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123. IEEE, 1990.

[17] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 73–83. IEEE, 2001.

[18] RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification, 2012.

[19] Zaid Al-bayati, Youcheng Sun, Haibo Zeng, Marco Di Natale, Qi Zhu, and Brett Meyer. Task placement and selection of data consistency mechanisms for real-time multicore applications. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 172–181. IEEE, 2015.

[20] Zaid Al-bayati, Qingling Zhao, Ahmed Youssef, Haibo Zeng, and Zonghua Gu. Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms. In *The 20th Asia and South Pacific Design Automation Conference*, pages 630–635. IEEE, 2015.

[21] Zaid Al-bayati, Jonah Caplan, Brett H Meyer, and Haibo Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 97–102. IEEE, 2016.

[22] Zaid Al-bayati, Brett H Meyer, and Haibo Zeng. Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2016 IEEE International Symposium on*, pages 57–62. IEEE, 2016.

[23] Qingling Zhao, Zaid Al-Bayati, Zonghua Gu, and Haibo Zeng. Optimized Implementation of Multirate Mixed-Criticality Synchronous Reactive Models. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):23:1–23:25, December 2016. ISSN 1084-4309.

[24] Michael R Garey and David S Johnson. Computers and intractability: a guide to the theory of NP-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[25] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[26] The AUTOSAR Standard, specification version 4.1, The AUTOSAR consortium, [Online] http://www.autosar.org.

[27] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[28] Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 9–pp. IEEE, 2005.

[29] Nathan Fisher, Sanjoy Baruah, and Theodore P Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10–pp. IEEE, 2006.

[30] Sanjoy Baruah and Enrico Bini. Partitioned scheduling of sporadic task systems: an ILP-based approach. In *Proceedings of the 2008 Conference on Design and Architectures for Signal and Image Processing*. Citeseer, 2008.

[31] Bipasa Chattopadhyay and Sanjoy Baruah. A lookup-table driven approach to partitioned scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 257–265. IEEE, 2011.

[32] Sanjoy Baruah. The partitioned EDF scheduling of sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 116–125. IEEE, 2011.

[33] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[34] Bjorn B Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 292–302. IEEE, 2013.

[35] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[36] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, et al. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Department of Computer Science, TU Dortmund, Tech. Rep*, 854, 2016.

[37] Theodore P Baker. A stack-based resource allocation policy for realtime processes. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 191–200. IEEE, 1990.

[38] Alexander Wieder and Bjorn B Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 45–56. IEEE, 2013.

[39] Jing Chen and Alan Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 236–246. IEEE, 1999.

[40] Hai Huang, Padmanabhan Pillai, and Kang G Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. *Ann Arbor*, 1001:48109–2122, 2002.

[41] Jing Chen and Alan Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS*, 1997.

[42] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Notices*, volume 25, pages 197–206. ACM, 1990.

[43] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–22. IEEE, 2010.

[44] Björn B Brandenburg. A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 196–206. IEEE, 2014.

[45] Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. *Proc. WMC, RTSS*, pages 1–6, 2013.

[46] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43. IEEE, 2011.

[47] Malcolm S Mollison, Jeremy P Erickson, James H Anderson, Sanjoy K Baruah, John Scoredos, et al. Mixed-criticality real-time scheduling for multicore systems. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1864–1871. IEEE, 2010.

[48] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50 (1):142–177, 2014.

[49] Paul Rodriguez, Laurent George, Yasmina Abdeddaïm, and Joël Goossens. Multi-criteria evaluation of partitioned edf-vd for mixed-criticality systems upon identical processors. In *Workshop on Mixed Criticality Systems*, 2013.

[50] Sanjoy K Baruah, Vincenzo Bonifaci, Gianlorenzo DAngelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *Algorithms–ESA 2011*, pages 555–566. Springer, 2011.

[51] Chuancai Gu, Nan Guan, Qingxu Deng, and Wang Yi. Partitioned mixed-criticality scheduling on multiprocessor platforms. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[52] Owen R Kelly, Hakan Aydin, and Baoxian Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1051–1059. IEEE, 2011.

[53] Pengcheng Huang, Hoeseok Yang, and Lothar Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *ACM Design Automation Conference*, 2014.

[54] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., 1st edition, 2007.

[55] Cristian Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium, 2008. RAMS 2008. Annual*, pages 370–374. IEEE, 2008.

[56] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Computing Surveys (CSUR)*, 46(1):8, 2013.

[57] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005.

[58] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Ieee Micro*, 25 (6):10–16, 2005.

[59] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 170–177. ACM, 2003.

[60] Gaurang Upasani, Xavier Vera, and Antonio González. Avoiding core's DUE & SDC via acoustic wave detectors and tailored error containment and recovery. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 37–48. IEEE Press, 2014.

[61] Infineon. AURIX Family - TC29xT, 2014. URL `http://www.infineon.com`.

[62] Freescale. MPC577xK 32-bit MCU for ADAS Applications, 2014. URL `http://www.freescale.com`.

[63] Renesas Electronics. RH850/P1x, 2016. URL `http://www.renesas.com`.

[64] Carles Hernandez and Jaume Abella. Live: Timely error detection in light-lockstep safety critical systems. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[65] B. H. Meyer, B. H. Calhoun, *et al.* Cost-effective safety and fault localization using distributed temporal redundancy. In *International Conference on Compilers, architecture and synthesis for embedded systems*, 2011.

[66] Jian Fu, Qiang Yang, Raphael Poss, Chris R Jesshope, and Chunyuan Zhang. On-demand thread-level fault detection in a concurrent programming environment. In *Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2013.

[67] J Lin, Albert MK Cheng, Douglas Steel, and Michael Yu-Chi Wu. Scheduling mixed-criticality realtime tasks with fault tolerance. In *2nd Workshop on Mixed Criticality Systems, RTSS*, 2014.

[68] RisatMahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4):509–547, 2014.

[69] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering*, 22(3):399–436, 2015.

[70] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *International Workshop on Embedded Software*, pages 290–305. Springer, 2003.

[71] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.

[72] C. Bolchini and A. Miele. Reliability-Driven System-Level Synthesis for Mixed-Critical Embedded Systems. *Computers, IEEE Transactions on*, 62(12):2489–2502, Dec 2013. ISSN 0018-9340. doi: 10.1109/TC.2012.226.

[73] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Mixed criticality scheduling in fault-tolerant distributed real-time systems. In *International Conference on Embedded Systems*, 2014.

[74] Shin-haeng Kang, Hoeseok Yang, Sungchan Kim, Iuliana Bacivarov, Soonhoi Ha, and Lothar Thiele. Static Mapping of Mixed-Critical Applications for Fault-Tolerant MPSoCs. In *ACM Design Automation Conference*, 2014.

[75] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000.

[76] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. *ACM SIGBED Review*, 6(3):6, 2009.

[77] Guangdong Liu, Ying Lu, and Shige Wang. An Efficient Fault Recovery Algorithm in Multiprocessor Mixed-Criticality Systems. In *High Performance Computing and Communications & IEEE International Conference on Embedded and Ubiquitous Computing, 2013*, pages 2006–2013. IEEE, 2013.

[78] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 291–300. IEEE, 2009.

[79] Gabriela Nicolescu and Pieter J Mosterman. *Model-based design for embedded systems*. CRC Press, 2009.

[80] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152.

[81] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[82] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[83] MathWorks. The Mathworks Simulink and StateFlow User's Manuals. web page: http://www.mathworks.com.

[84] Zaid Al-bayati, Haibo Zeng, Marco Di Natale, and Zonghua Gu. Multitask implementation of synchronous reactive models with earliest deadline first scheduling. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 168–177. IEEE, 2013.

[85] Juan Pablo Vielma, Shabbir Ahmed, and George Nemhauser. Mixed-integer models for nonseparable piecewise-linear optimization: Unifying framework and extensions. *Operations research*, 58(2):303–315, 2010.

[86] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 7 (2):15, 2008.

[87] Marco Di Natale and Valerio Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–32, 2008.

[88] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 3–12. IEEE, 2011.

[89] Sanjoy Baruah. Implementing mixed-criticality synchronous reactive programs upon uniprocessor platforms. *Real-Time Systems*, 50(3):317–341, 2014.

[90] Alexander Wieder and Bjorn B Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 49–58. IEEE, 2013.

[91] Farhang Nemati, Thomas Nolte, and Moris Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *Principles of Distributed Systems*, pages 253–269. Springer, 2010.

[92] IBM ILOG CPLEX Optimization Studio. URL `https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[93] Eugene Yip, Matthew MY Kuo, Partha S Roop, and David Broman. Relaxing the synchronous approach for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 89–100. IEEE, 2014.

[94] Giorgio C Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic task model for adaptive rate control. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 286–295. IEEE, 1998.

[95] Hang Su and Dakai Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 147–152. EDA Consortium, 2013.

[96] Hang Su, Dakai Zhu, and Daniel Mossé. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 352–357. IEEE, 2013.

[97] Hang Su, Nan Guan, and Dakai Zhu. Service guarantee exploration for mixed-criticality systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE International Conference on*, pages 1–10. IEEE, 2014.

[98] Dionisio De Niz, Lutz Wrage, Nathaniel Storer, Anthony Rowe, and Ragunathan Raj Rajkumar. On resource overbooking in an unmanned aerial vehicle. In *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*, pages 97–106. IEEE Computer Society, 2012.

[99] Dionisio De Niz, Lutz Wrage, Anthony Rowe, and Ragunathan Raj Rajkumar. Utility-based resource overbooking for cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s): 162, 2014.

[100] Hang Su, Peng Deng, Dakai Zhu, and Qi Zhu. Fixed-Priority Dual-Rate Mixed-Criticality Systems: Schedulability Analysis and Performance Optimization. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 59–68. IEEE, 2016.

[101] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[102] Petru Eles, Viacheslav Izosimov, Paul Pop, and Zebo Peng. Synthesis of fault-tolerant embedded systems. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1117–1122. ACM, 2008.

[103] JGAP. JGAP: Java Genetic Algorithms Package. `http://jgap.sourceforge.net`, 2012.

[104] Abhilash Thekkilakattil, Alan Burns, Radu Dobrin, and Sasikumar Punnekkat. Mixed Criticality Systems: Beyond Transient Faults. *3rd International Workshop on Mixed Criticality Systems, WMC 2015*.

[105] Neil C Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[106] Robert I Davis and Alan Burns. Robust priority assignment for fixed priority real-time systems. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 3–14. IEEE, 2007.

[107] Robert P Dick, David L Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.